A Brain-Friendly Guide

Head First Ajax

Fireside Chats



Learn how to make your web pages talk and listen at the same time



Transfer your data with plain text, XML, and JSON

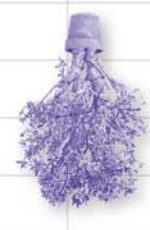




Learn how Sally did two things at the same time with asynchronous programming



Make your clunky web apps feel like dynamic, responsive desktop applications



Get a handle on trees and the Document Object Model

O'REILLY®

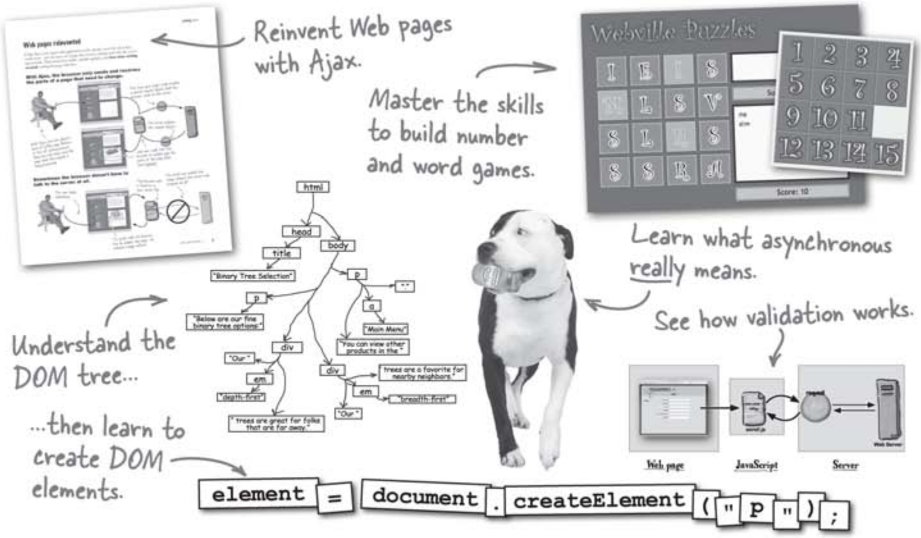
Rebecca M. Riordan

Head First Ajax

Ajax/Web Programming

What will you learn from this book?

Head First Ajax is a complete learning experience for dynamic, interactive web application programming. Built for your brain, this book covers JavaScript, XHTML, asynchronous and synchronous requests, the DOM, and everything else you'll need to expand and revolutionize the abilities of your web apps. You'll do more than just memorize methods from a framework someone else wrote, or drag and drop widgets from a toolkit. By the time you're through, you'll be building apps that talk to a server without page reloads, move items around on the screen in response to users' actions, and even anticipate your users' mistakes…before they're made.



Why does this book look so different?

We think your time is too valuable to spend struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, *Head First Ajax* uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.

"You don't just read Head First books, you *do* Head First books. And it makes all the difference."

—*Pauline McNamara*
Center for New Technologies and Education, Fribourg University, Switzerland

"Behind the Ajax eight ball? Get out of the shadows with this book. You'll wrap your mind around the core concepts, and have some fun in the process."

—*Bear Bibeault*
Web Applications Architect

US $44.99 CAN $44.99
ISBN: 978-0-596-51578-2

Safari Books Online
Free online edition for 45 days with purchase of this book. Details on last page.

O'REILLY®
www.oreilly.com
www.headfirstlabs.com

Advance Praise for Head First Ajax

"Ajax is more than just revisiting existing technologies, making some small changes to your web application and then declaring it Ajax-enabled. Rebecca M. Riordan walks you through all of the steps of building an Ajax application in *Head First Ajax*, and shows you that Ajax is more than 'that little asynchronous part', but a better approach to web design all together."

— Anthony T. Holdener III, author of "Ajax: The Definitive Guide"

"You don't just read Head First books, you *do* Head First books. And it makes all the difference."

— Pauline McNamara, Techno-pedagogical Consultant for university eLearning projects, Switzerland

"The author does an excellent job teaching the various aspects of Ajax, bringing back previous lessons without being repetitive and introducing common problems in a way that helps readers discover the problems themselves. In areas where there still isn't a definitive practice, the reader is exposed to all the options and encouraged to make up his or her own mind."

— Elaine Nelson, Website Designer

"Behind the Ajax eight ball? Get out of the shadows with this book. You'll wrap your mind around the core concepts, and have some fun in the process."

— Bear Bibeault, Web Applications Architect

Other related books from O'Reilly

Ajax Design Patterns

Ajax: The Definitive Guide

Ajax Hacks™

Ajax on Java

Programming ASP.NET Ajax

JavaScript: The Definitive Guide

JavaScript: The Good Parts

Other books in O'Reilly's Head First series

Head First Java™

Head First Object-Oriented Analysis and Design (OOA&D)

Head First HTML with CSS and XHTML

Head First Design Patterns

Head First Servlets and JSP

Head First EJB

Head First PMP

Head First SQL

Head First C#

Head First Software Development

Head First JavaScript

Head First Physics

Head First Statistics

Head First Rails (2008)

Head First PHP & MySQL (2008)

Head First Web Design (2008)

Head First Algebra (2008)

Head First Ajax



Rebecca M. Riordan

O'REILLY®

Beijing • Cambridge • Köln • Sebastopol • Taipei • Tokyo

Head First Ajax

by Rebecca M. Riordan

Copyright © 2008 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators:	Kathy Sierra, Bert Bates
Series Editor:	Brett D. McLaughlin
Design Editor:	Louise Barr
Cover Designers:	Louise Barr, Steve Fehler
Production Editor:	Brittany Smith
Indexer:	Julie Hawks
Application Designer:	Fletcher Moore

Printing History:

August 2008: First Edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First Ajax*, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No web pages were harmed in the creation of this book, although several were bent significantly out of shape. Several browser-server communications were interrogated, but only after their rights were carefully presented.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-51578-2

[M] [12/08]

Author of Head First Ajax



Rebecca M. Riordan

With twenty years experience in the field, **Rebecca M. Riordan** has earned an international reputation for designing and implementing computer systems that are technically sound, reliable, and effectively meet her clients' needs. Her particular area of expertise is database design, and she is a five year Access MVP.

Rebecca has also authored several database books and served as a senior technical support engineer for Microsoft's database products. As a Microsoft employee in Australia, Rebecca was the Senior technical support engineer for Microsoft's database products.

Table of Contents (Summary)

	Intro	xix
1	Using Ajax: *Web Apps for a New Generation*	1
2	Designing Ajax Applications: *Thinking Ajaxian*	43
3	Javascript Events: *Reacting to your users*	93
4	Multiple Event Handlers: *Two's Company*	139
5	Asynchronous Applications: *It's Like Renewing Your Driver's License*	173
6	The Document Object Model: *Web Page Forestry*	229
7	Manipulating the DOM: *My Wish is Your Command*	283
8	Frameworks and Toolkits: *Trust No One*	329
9	XML Requests and Responses: *More Than Words Can Say*	343
10	JSON: *SON of JavaScript*	379
11	Forms and Validation: *Say What You Meant to Say*	407
12	Post Requests: *Paranoia: It's Your Friend*	445
i	Appendix i: *Top Five Topics We Didn't Cover*	471
ii	Appendix ii: *Utility Functions*	483

Table of Contents (the real thing)

Intro

Your brain on Ajax. Here *you* are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing Ajax?

Who is this book for?	xx
We know what you're thinking	xxi
Metacognition	xxiii
Bend your brain into submission	xxv
Read me	xxvi
The technical review team	xxviii
Acknowledgments	xxix

1 using ajax

Web Apps for a New Generation

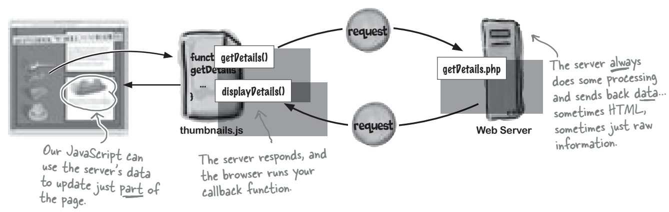
Tired of waiting around for your page to reload?

Frustrated by clunky web application interfaces? It's time to give your web apps that slick, responsive desktop feel. And how do you do that? With **Ajax**, your ticket to building Internet applications that are *more interactive, more responsive,* and *easier to use*. So skip your nap; it's time to put some polish on your web apps. It's time to get rid of unnecessary and slow full-page refreshes forever.

Web pages: the old-fashioned approach	2
Web pages reinvented	3
So what makes a page "Ajax" ?	5
Rob's Rock 'n' Roll Memorabilia	6
Ajax and rock 'n' roll in 5 steps	12
Step 1: Modify the XHTML	14
Step 2: Initialize the JavaScript	16
Step 3: Create a request object	20
Step 4: Get the item's details	22
Let's write the code for requesting an item's details	24
Always make sure you have a request object before working with it	25
The request object is just an object	26
Hey, server... will you call me back at displayDetails(), please?	27
Use send() to send your request	28
The server usually returns data to Ajax requests	30
Ajax is server-agnostic	31
Use a callback function to work with data the server returns	35
Get the server's response from the request object's responseText property	36
Goodbye traditional web apps...	38



Ajax pages only talk to the server when they have to... and only about what the server knows.

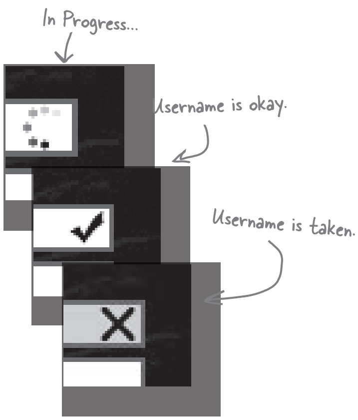


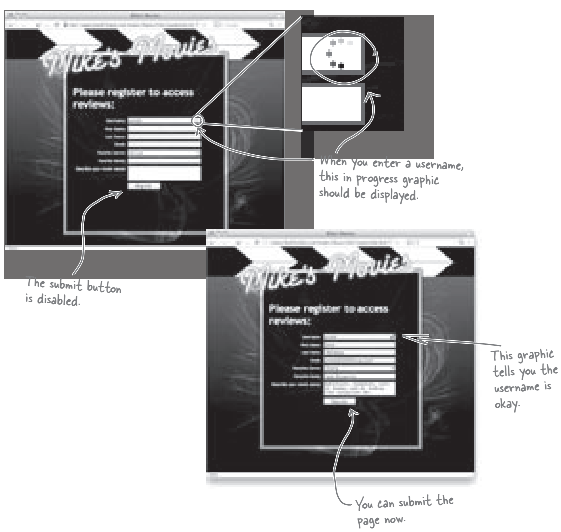
2 designing ajax applications

Thinking Ajaxian

Welcome to Ajax apps—it's a whole new web world.

So you've built your first Ajax app, and you're already thinking about how to change all your web apps to make requests asynchronously. But that's not all there is to Ajax programming. You've got to ***think about your applications differently***. Just because you're making asynchronous requests, doesn't mean your application is user-friendly. It's up to you to help your users **avoid making mistakes**, and that means **rethinking** your entire application's **design**.





Mike's traditional web site sucks	44
Let's use Ajax to send registration requests ASYNCHRONOUSLY	46
Update the registration page	51
Event Handlers Exposed	53
Set the window.onload event handler... PROGRAMMATICALLY	54
Code in your JavaScript outside of functions runs when the script is read	56
What happens when...	57
And on the server...	58
Some parts of your Ajax designs will be the same... every time	60
createRequest() is always the same	61
Create a request object... on multiple browsers	64
Ajax app design involves both the web page AND the server-side program	66
The request object connects your code to the web browser	72
You talk to the browser, not the server	73
The browser calls back your function with the server's response	76
Show the Ajax registration page to Mike...	78
The web form has TWO ways to send requests to the server now	79
Let's create CSS classes for each state of the processing...	82
...and change the CSS class with our JavaScript	83
Changes? We don't need no stinkin' changes!	84
Only allow registration when it's appropriate	85

3 javascript events

Reacting to your users

Sometimes you need your code to react to other things going on in your web application... and that's where **events** come into the picture. An event is *something that happens* on your page, in the browser, or even on a web server. But it's not enough to just know about events... sometimes you want to respond to them. By creating code, and registering it as an **event handler**, you can get the browser to run your handler every time a particular event occurs. Combine events and handlers, and you get **interactive web applications**.

It all started with a downward-facing dog... 94

Ajax apps are more than the sum of their parts 101

Here's Marcy's XHTML... 102

Events are the key to interactivity 104

Connect events on your web page to event handlers in your JavaScript 107

Use window.onload event to initialize the interactivity on a web page 108

Change those left-side images to be clickable 113

Use your XHTML's content and structure 114

Add the code for hideHint(), too 117

Tabs: an optical (and graphical) illusion 118

Use a for... loop to cycle through the images 119

CSS classes are the key (again) 120

Ummm... but the tabs aren't <a>'s ! 121

This broke our JavaScript, too, didn't it? 122

Use a request object to fetch the class details from the server 127

Two functions changing the same part of a web page 128

When you need to change images in your script, think "change CSS classes" 133

Links in XHTML are represented by <a> elements 134

We need a function to show an active button and hide a button, too 135







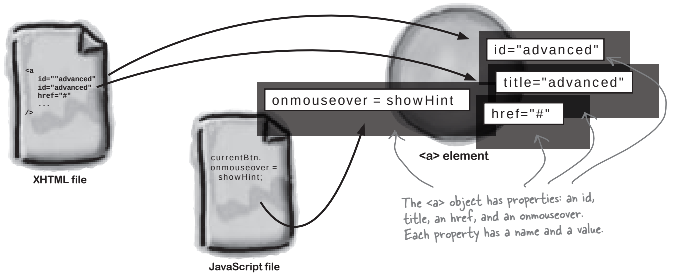
4 multiple event handlers

Two's company

A single event handler isn't always enough.

Sometimes you've got more than one event handler that needs to be called by an event. Maybe you've got some event-specific actions, as well as some generic code, and stuffing everything into a single event handler function won't cut it. Or maybe you're just trying to build **clean, reusable code**, and you've got **two bits of functionality** triggered by the **same event**. Fortunately, we can use some **DOM Level 2** methods to assign multiple handler functions to a **single event**.

An event can have only one event handler attached to it (or so it seems)	140
Event handlers are just properties	141
A property can have only ONE value	141
Assign multiple event handlers with addEventListener()	142
Objects can have multiple event handlers assigned to a single event in DOM Level 2	144
What's going on with Internet Explorer?	148
Internet Explorer uses a totally different event model	149
attachEvent() and addEventListener() are functionally equivalent	149
addEventHandler() works for ALL apps, not just Marcy's yoga page	154
Let's update initPage() to use our new utility function	155
Use an alert() to troubleshoot	157
So what else could be going wrong?	157
Event handlers in IE are owned by IE's event framework	159
attachEvent() and addEventListener() supply another argument to our handlers	160
We need to name the Event argument, so our handlers can work with it	161
You say target, I say srcElement...	162
So how do we actually GET the object that triggered the event?	166



5 asynchronous applications

It's like renewing your driver's license

Are you tired of waiting around? Do you hate long delays? You can do something about it with asynchrony!

You've already built a couple of pages that made asynchronous requests to the server to avoid making the user sit around waiting for a page refresh. In this chapter, we'll dive even deeper into the details of building asynchronous applications. You'll find out what **asynchronous really means**, learn how to use **multiple asynchronous requests**, and even build a **monitor function** to keep all that asynchrony from confusing you and your users.

What does asynchronous really mean?	174
You've been building asynchronous apps all along	176
But sometimes you barely even notice...	177
Speaking of more server-side processing...	178
(More) Asynchrony in 3 easy steps	181
We need two password fields and a <div> for the cover images	182
If you need new behavior, you probably need a new event handler function	187
With ONE request object, send and receive ONE asynchronous request	196
Asynchronous requests don't wait on anything... including themselves!	197
If you're making TWO separate requests, use TWO separate request objects	198
You can't count on the ORDERING of your requests and responses	204
A monitor function monitors your application from OUTSIDE the action	209
You call a monitor function when action MIGHT need to be taken	210
Status variables let monitors know what's going on	212
And now for our last trick...	216
Synchronous requests block ALL YOUR CODE from doing anything	218
Use setInterval() to let JavaScript run your process, instead of your own code	221

An asynchronous request for cola

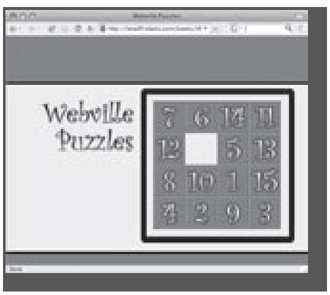


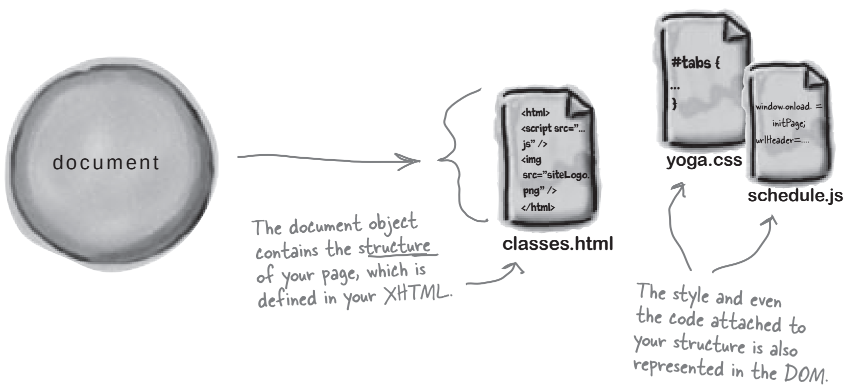
6 the document object model

Web Page Forestry

Wanted: easy-to-update web pages. It's time to take things into your own hands and start writing code that updates your web pages on the fly. Using the **Document Object Model**, your pages can take on new life, responding to users' actions, and you can ditch unnecessary page reloads forever. By the time you've finished this chapter, you'll be able to find, move, and update content virtually anywhere on your web page.

You can change the CONTENT of a page...	230
...or you can change the STRUCTURE of a page	231
Browsers use the Document Object Model to represent your page	232
Here's the XHTML that you write...	234
...and here's what your browser sees	235
Your page is a set of related objects	237
Let's use the DOM to build a dynamic app	244
You start with XHTML...	246
appendChild() adds a new child to a node	255
You can locate elements by name or by id	256
Interview with a new parent	259
Can I move the clicked tile?	260
You can move around a DOM tree using FAMILY relationships	262
A DOM tree has nodes for EVERYTHING in your web page	272
The nodeName of a text node is "#text"	274
Did I win? Did I win?	278
But seriously... did I win?	279



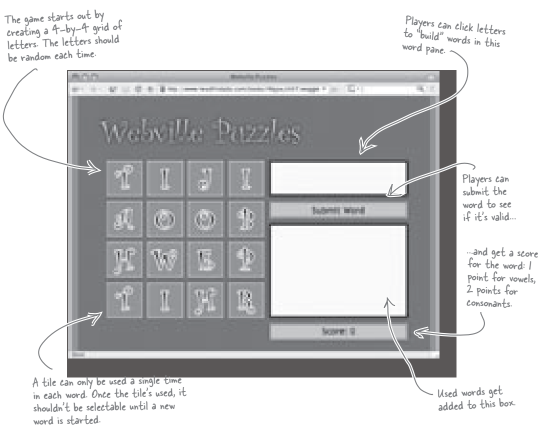


7 manipulating the DOM

My wish is your command

Sometimes you just need a little ~~mind~~ DOM control.

It's great to know that web browsers turn your XHTML into DOM trees. And you can do a lot by moving around within those trees. But real power is **taking control of a DOM tree** and making the tree look like *you* want it to. Sometimes what you really need is to **add a new element** and some text, or to **remove an element**, like an <img>, from a page altogether. You can do all of that and more with the DOM, and along the way, **banish that troublesome innerHTML property** altogether. The result? Code that can do more to a page, *without* having to mix presentation and structure in with your JavaScript.



Webville Puzzles... the franchise 284
Woggle doesn't use table cells for the tiles 288
The tiles in the XHTML are CSS-positioned 289
"We don't want TOTALLY random letters..." 291
Our presentation is ALL in our CSS 293
We need a new event handler for handling tile clicks 295
Start building the event handler for each tile click 296
We can assign an event handler in our randomizeTiles() function 296
Property values are just strings in JavaScript 297
We need to add content AND structure to the "currentWord" div 300
Use the DOM to change a page's structure 300
Use createElement() to create a DOM element 301
You have to TELL the browser where to put any new DOM nodes you create 302
We need to disable each tile. That means changing the tile's CSS class... 310
...AND turning OFF the addLetter() event handler 310
Submitting a word is just (another) request 312
Our JavaScript doesn't care how the server figures out its response to our request 312
Usability check: WHEN can submitWord() get called? 317



8 frameworks and toolkits

Trust No One

So what's the *real* story behind all those Ajax frameworks?

If you've been in Webville awhile, you've probably run across at least one JavaScript or Ajax framework. Some frameworks give you **convenience methods for working with the DOM**. Others make **validation** and **sending requests** simple. Still others come with libraries of pre-packaged JavaScript **screen effects**. But which one should you use? And how do you know what's really going on inside that framework? It's time to do more than use other people's code... it's time to ***take control of your applications***.

Reasons to use a framework

Reasons NOT to use a framework

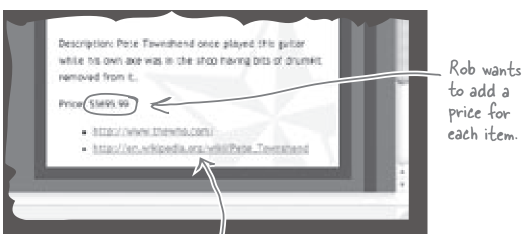
So what frameworks ARE there?	335
Every framework uses a different syntax to do things	336
The syntax may change... but the JavaScript is still the same	337
To framework or not to framework?	340
The choice is up to you...	342

9 xml requests and responses

More Than Words Can Say

How will you describe yourself in 10 years? How about 20?

Sometimes you need **data that can change with your needs**... or the needs of your customers. Data you're using now might need to change in a few hours, or a few days, or a few months. With XML, the *extensible markup language*, your data can **describe itself**. That means your scripts won't be filled with ifs, elses, and switches. Instead, you can use the descriptions that XML provides about itself to figure out how to ***use*** the data the XML contains. The result: **more flexibility** and **easier data handling**.



Classic rock gets a 21st century makeover	344
How should a server send a MULTI-valued response?	347
innerHTML is only simple for the CLIENT side of a web app	353
You use the DOM to work with XML, just like you did with XHTML	359
XML is self-describing	366

json

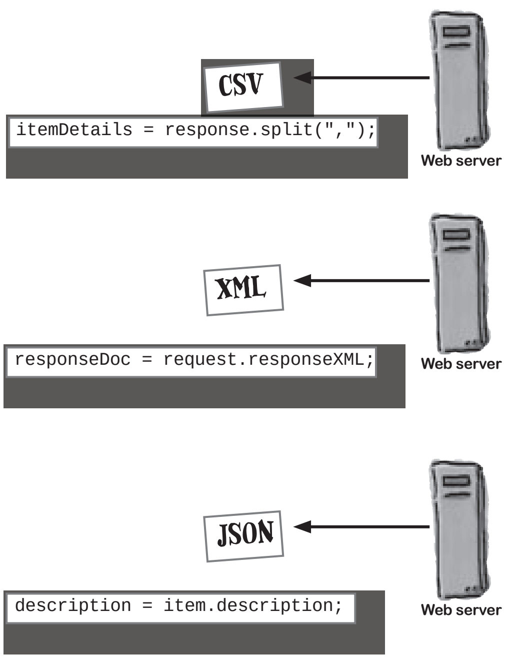
10 SON of JavaScript

JavaScript, objects, and notation, oh my!

If you ever need to represent objects in your JavaScript, then you're going to love JSON, **JavaScript Standard Object Notation**. With JSON, you can **represent complex objects and mappings** with text and a few curly braces. Even better, you can **send and receive JSON** from other languages, like PHP, C#, Python, and Ruby.

JSON can be text AND an object	381
JSON data can be treated as a JavaScript object	382
So how do we get JSON data from the server's response?	383
JavaScript can evaluate textual data	385
Use eval() to manually evaluate text	385
Evaluating JSON data returns an object representation of that data	386
JavaScript objects are already dynamic because they're not compiled objects	392
Access an object's members, then get an object's values with those members	393
You need to PARSE the server's response, not just EVALUATE it	399

JSON can be text AND an object



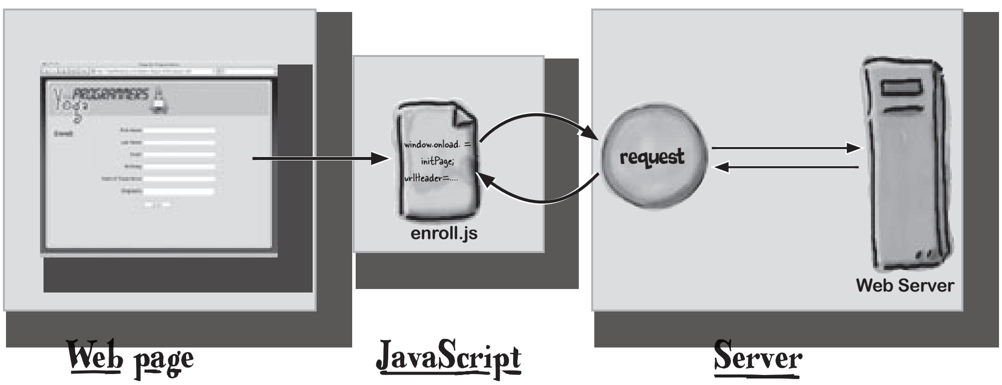
11 forms and validation

Say what you meant to say

Everyone makes mistakes from time to time.

Give a human being a chance to talk (or type) for a few minutes, and they'll probably make at least one or two **mistakes**. So how do your web apps **respond to those mistakes**? You've got to **validate** your users' input and react when that input has problems. But who does what? What should your web page do? What should your JavaScript do? And what's the role of the server in **validation** and **data integrity**?

Validation should work from the web page BACK to the server	414
You can validate the FORMAT and CONTENT of data	420
Don't Repeat Yourself: DRY	423
Let's build some more event handlers	426
RETURN of SON of JavaScript	430
The value of a property can be another JavaScript object	430
Let's warn Marcy's customers when there's a problem with their entry	433
If you don't warn(), you have to unwarn()	437
IF there's a warning, get rid of it	437
Duplicate data is a SERVER problem	443



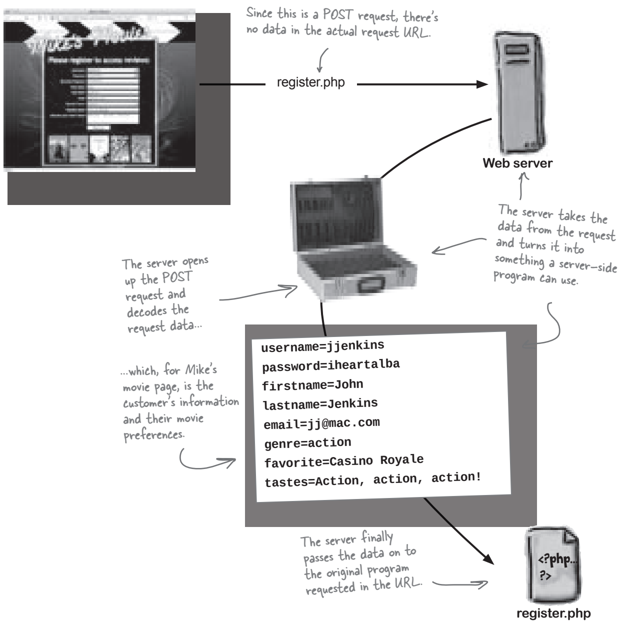
12 post requests

Paranoia: It's your friend

Someone's watching you. Right now. Seriously.

Freedom of Information Act? Isn't that called the Internet? These days, anything a user types into a form or clicks on a web page is subject to **inspection**. Whether it's a network admin, a software company trying to learn about your trends, or a malicious hacker or spammer, your ***information isn't safe unless you make it safe***. When it comes to web pages, you've got to **protect your users' data** when they click Submit.

GET requests send request parameters across the network as clear text	449
POST requests DON'T send clear text	450
The data in a POST request is ENCODED until it reaches the server	452
send() your request data in a POST request	454
Always check to make sure your request data was RECEIVED.	456
Why didn't the POST request work?	458
The server unencodes POST data	459
We need to TELL the server what we're sending	460
Set a request header using setRequestHeader() on your request object	462

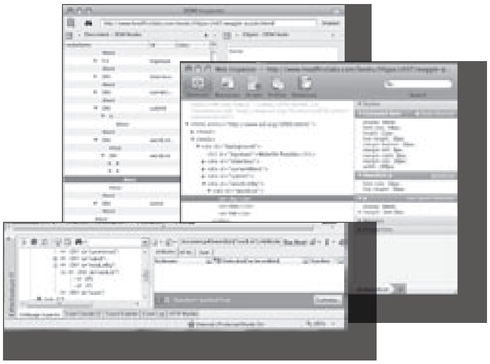


leftovers

The Top Five Topics (we didn't cover)

It's been a long ride... and you're almost to the end.

We can barely stand to let you go, but before you do, there's still a few things left to cover. We can't possibly fit everything about Ajax into one 600-page book. So we threw out everything you didn't really need to know and kept the last few important bits in this appendix.



#1 Inspecting the DOM 472

#2 Graceful degradation 475

#3 script.aculo.us and the Yahoo UI libraries 476

#4 Using JSON libraries in your PHP code 478

#5 Ajax and ASP.NET 480

utility functions

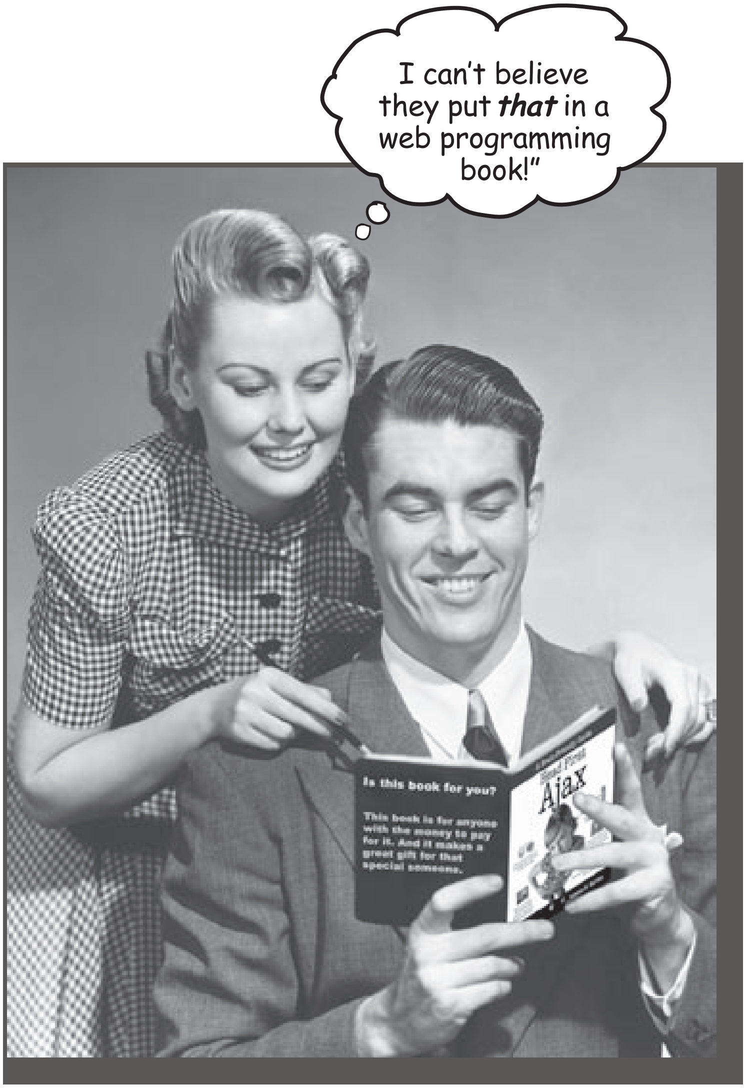
Just Gimme the Code

Sometimes you just want everything in one place.

You've spent a lot of time using utils.js, our little utility class of Ajax, DOM, and event utility functions. Inside these pages, you'll get all those functions in one place, ready to put into your own utility scripts and applications.

utils.js: a work in progress 484

Intro



In this section we answer the burning question: "So why DID they put that in a web programming book?"

Who is this book for?

If you can answer "yes" to all of these:

1. Do you know **HTML**, some **CSS**, and some **JavaScript**? (You don't need to be a guru or anything.)
2. Do you want to **learn**, **understand**, and ***remember*** Ajax, with a goal of developing responsive and usable web applications?
3. Do you prefer **stimulating dinner party conversation** to **dry**, **dull**, **academic lectures**?

this book is for you.

Who should probably back away from this book?

If you can answer "yes" to any ***one*** of these:

1. Are you **completey new** to HTML or CSS or JavaScript? (You don't need to be advanced, but you should definitely have some experience. If not, go get a copy of *Head First HTML and CSS*, today, and then come back and get this book.)
2. Are you a kick-butt Ajax or web developer looking for a ***reference* book**?
3. Are you **afraid to try something different**? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if servers and web browsers are anthropomorphized?

this book is not for you.



[Note from marketing: this book is for anyone with a credit card.]

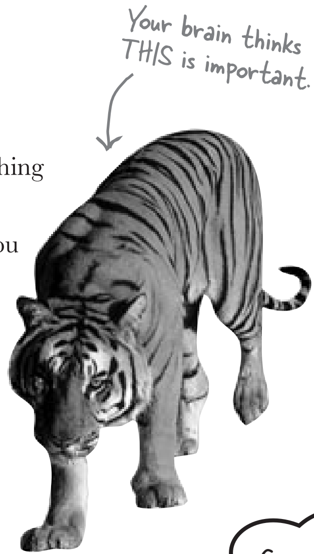
We know what you're thinking

"How can *this* be a serious book on web programming?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

We know what your *brain* is thinking



Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

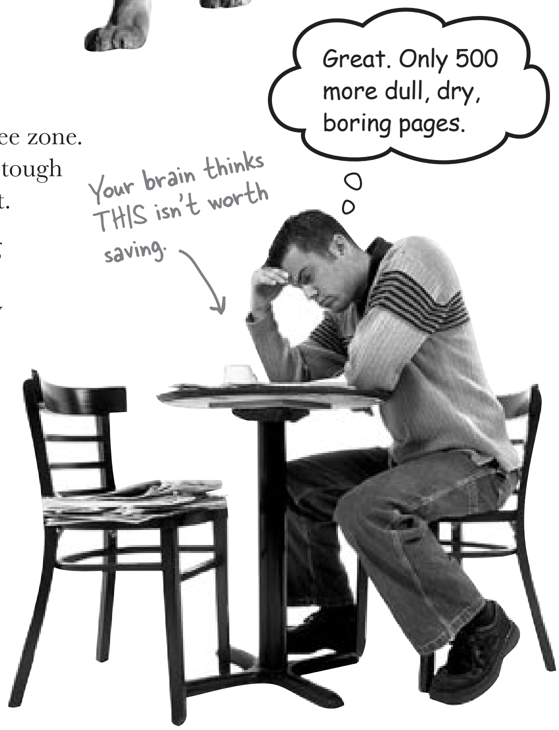
How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you, what happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge*.

And that's how your brain knows...

This must be important! Don't forget it!

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.



Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* non-important content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those "party" photos on your Facebook page.

And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."

We think of a "Head First" reader as a learner.

So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

Some of the Head First learning principles:

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. Put the words within or near the graphics they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.



Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner party companion, or a lecture?

Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.



Get—and keep—the reader's attention. We've all had the "I really want to learn this but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the...?" , and the feeling of "I Rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm a more senior programmer than you" Bob from the next cube over *doesn't*.

This time, your page's code creates a special request object that the browser sends to the server.



Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us didn't take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn Ajax and web programming. And you probably don't want to spend a lot of time. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.



The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to treat web programming like it was a hungry tiger?

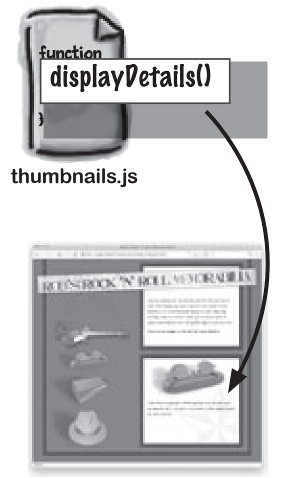
There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to makes sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning…

Here's what WE did:



We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.



We included more than 80 ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the exercises challenging-yet-do-able, because that's what most people prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.



We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

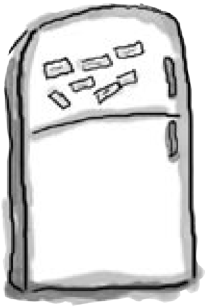
And we included ***stories*** and exercises that present ***more than one point of view,*** because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.



We included ***challenges***, with exercises, and by asking ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, etc., because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.





Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

1. **Slow down. The more you understand, the less you have to memorize.**

   Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2. **Do the exercises. Write your own notes.**

   We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3. **Read the "There are No Dumb Questions"**

   That means all of them. They're not optional sidebars—***they're part of the core content!*** Don't skip them.

4. **Make this the last thing you read before bed. Or at least the last challenging thing.**

   Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5. **Drink water. Lots of it.**

   Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

6. **Talk about it. Out loud.**

   Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

7. **Listen to your brain.**

   Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8. **Feel something.**

   Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9. **Practice writing web applications!**

   There's only one way to truly master web programming: **program web applications**. And that's what you're going to do throughout this book. Using Ajax is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has apps that we'll build. Don't just skip over them—a lot of the learning happens when you build these apps yourself. And definitely make sure you understand what's going on before you move on to the next part of the book.

Read Me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We assume you are familiar with HTML and CSS.

It would take an entire book to teach you HTML and CSS (in fact, that's exactly what it took: *Head First HTML with CSS & XHTML*). We chose to focus this book on Ajax programming, rather than rehash lots of markup and style that you could learn about in other places.

We assume you've at least seen JavaScript code before.

It would take an entire book to teach you... oh, wait, we've already said that. Seriously, JavaScript is a lot more than a simple scripting language, and we aren't going to cover all the ways you can use JavaScript in this book. You'll learn about all the ways that JavaScript is related to Ajax programming, and learn how to use JavaScript extensively to add interaction to your web pages and make requests to a server.

However, if you've never written a line of JavaScript, aren't at all familiar with functions or curly braces, or have never programmed in any language before, you might want to pick up a good JavaScript book and browse through it. If you want to plow into this book, feel free—but we will be moving fairly quickly over the basics.

We don't cover server-side programming in this book.

It's now common to find server-side programs written in Java, PHP, Ruby, Python, Perl, Ruby on Rails, C#, and a whole lot more. Ajax programming works with all of these languages, and we have tried to represent several of them in this book's examples.

To keep you focused on learning Ajax, though, we do not spend much time explaining the server-side programs used; we'll show you the basic inputs and outputs to the server, but that's as far as we go. We believe that your Ajax applications can be written to work with any kind of server-side program; we also believe that you're smart enough to apply the lessons learned from an example that uses PHP to one that uses Ruby on Rails or a Java servlet.

You can visit us online at **http://www.headfirstlabs.com/books/hfajax** to download sample server-side programs, so you can run these apps yourself.

We encourage you to use more than one browser with this book.

As much as it sucks, different web browsers handle your HTML, your CSS, and your JavaScript in completely different ways. If you want to be a complete Ajax programmer, you should always test your asynchronous applications on lots of modern browsers. All the examples in this book were tested on recent versions of Firefox, Opera, Safari, Internet Explorer, and Mozilla. If you find problems, though, let us know... we promise it's an accident.

We often use tag names for element names.

Rather than saying "the **a** element," or "the 'a' element," we use a tag name, like "the **<a>** element." While this may not be technically correct (because **<a>** is an opening tag, not a full blown element), it does make the text more readable.

The activities are NOT optional.

The exercises and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. ***Don't skip the exercises.***

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about learning, so you'll see many of the concepts come up more than once.

The examples are as lean as possible.

Our readers tell us that it's frustrating to wade through 200 lines of an example looking for the two lines they need to understand. Most examples in this book are shown within the smallest possible context, so that the part you're trying to learn is clear and simple. Don't expect all of the examples to be robust, or even complete—they are written specifically for learning, and aren't always fully functional.

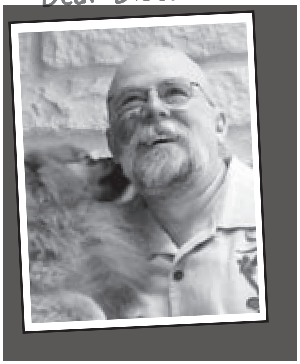
We've placed all the example files on the Web so you can download them. You'll find them at **`http://www.headfirstlabs.com/books/hfajax/`**.

The 'Brain Power' exercises don't have answers.

For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises you will find hints to point you in the right direction.

The technical review team

Bear Bibeault



Anthony T. Holdener III



Elaine Nelson



Not pictured (but just as awesome): Chris Haddix and Stephen Tallent

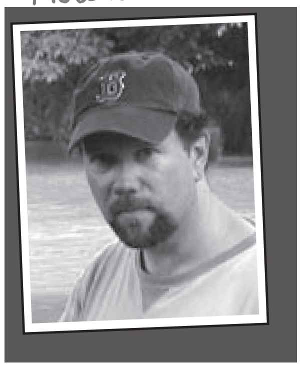
Pauline McNamara



Andrew Monkhouse



Fletcher Moore



Technical Reviewers:

Bear Bibeault is a Web Applications Architect responsible for an enterprise financial application used by the accountants that many of the Fortune 500 companies keep in their dungeons. He also delights clients with web applications he creates on the side (Ajax-powered, of course), and serves as a sheriff (senior moderator) at JavaRanch.com.

Anthony T. Holdener III is the Director of Information Technology for Korein Tillery, LLC, but was a web applications developer in his previous life as a programmer; he is also the author of *Ajax: The Definitive Guide* (O'Reilly).

Elaine Nelson has been designing websites for nearly 10 years. As she tells her mother, an English degree comes in handy everywhere. Elaine's current musings and obsessions can be found at elainenelson.org.

Pauline McNamara has worked with university e-Learning projects in Switzerland for the past 6 years. Her current learning passion involves raising an adorable puppy with her partner (who's also adorable).

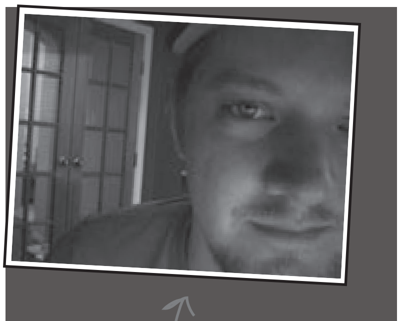
Andrew Monkhouse is an administrator on JavaRanch, and a Java developer by day. He is currently working for PersonalShopper.com in the USA - a long way from his home in Australia.

Fletcher Moore did all our code samples and was indispensable to the project. He is a web developer and designer at Georgia Tech. In his spare time he's an avid cyclist, musician, gardener, and Red Sox fan. He resides in Atlanta with his wife Katherine, daughter Sailor, and son Satchel.

Acknowledgments

My support team:

Even beyond the fantastic technical reviewers, this book wouldn't have seen the light of day were it not for my own little technical support team. **Stephen Jeffries**, developer extraordinaire, helped out with the server-side code and reviewed all the samples before submission. **Michael Morrison** is the world's best sounding board for those middle-of-the-night, "Why does this code work everywhere but in this browser?" questions. And **John Hardesty** provided the basic algorithms for several of the games you'll be building. (He's also my brother, which I've always thought was pretty cool. But don't tell him I said so.)



Brett McLaughlin

My editor:

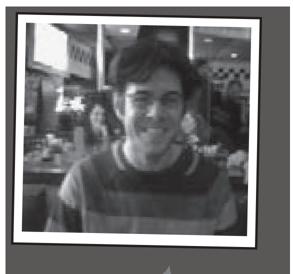
The greatest gift any editor can give a writer is to say, "This isn't working, this is why, and this is what I think you should do about it." **Brett McLaughlin** came through every damn time, and for that I will always be grateful.

The O'Reilly team:

I'd also like to thank **Laurie Petrycki** for being the voice of reason in the always-insane world of technical publishing, and both **Louise Barr** and **Sanders Kleinfeld** for fielding even the wildest InDesign questions with patience and grace.



Lou Barr



Sanders Kleinfeld

Safari® Books Online

When you see a Safari® icon on the cover of your favorite technology book that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at `http://safari.oreilly.com`.

1 using ajax

Web Apps for a New Generation

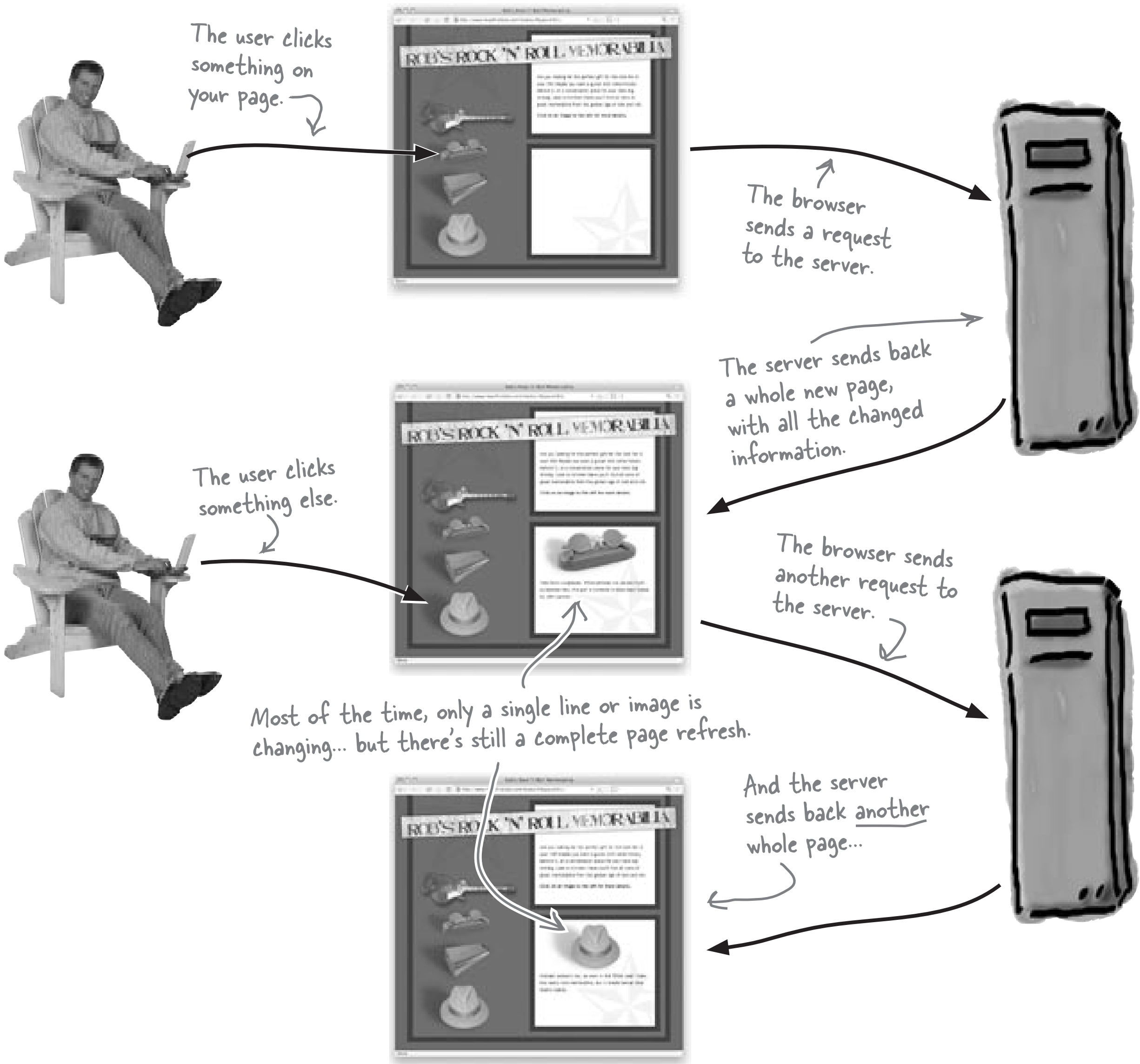


Tired of waiting around for your page to reload?

Frustrated by clunky web application interfaces? It's time to give your web apps that slick, responsive desktop feel. And how do you do that? With **Ajax**, your ticket to building Internet applications that are *more interactive, more responsive,* and *easier to use*. So skip your nap; it's time to put some polish on your web apps. It's time to get rid of unnecessary and slow full-page refreshes forever.

Web pages: the old-fashioned approach

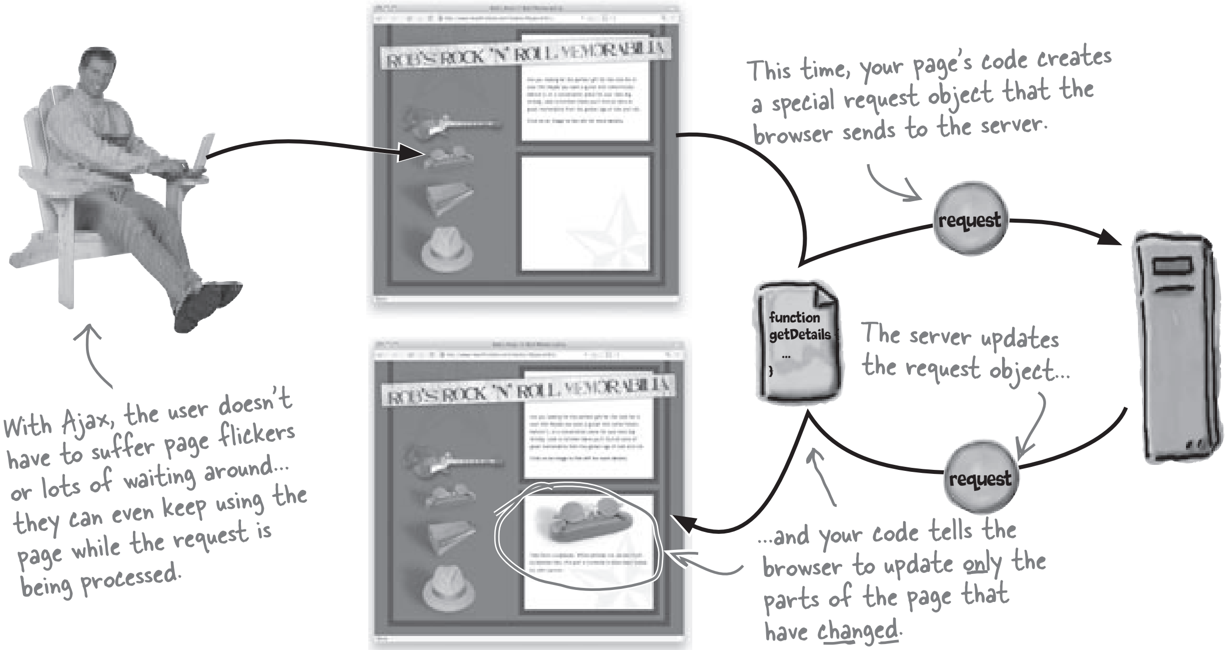
With traditional web pages and applications, every time a user clicks on something, the browser sends a request to the server, and the server responds with a **whole new page**. Even if your user's web browser is smart about caching things like images and cascading style sheets, that's a lot of traffic going back and forth between their browser and your server... and a lot of time that the user sits around waiting for full page refreshes.



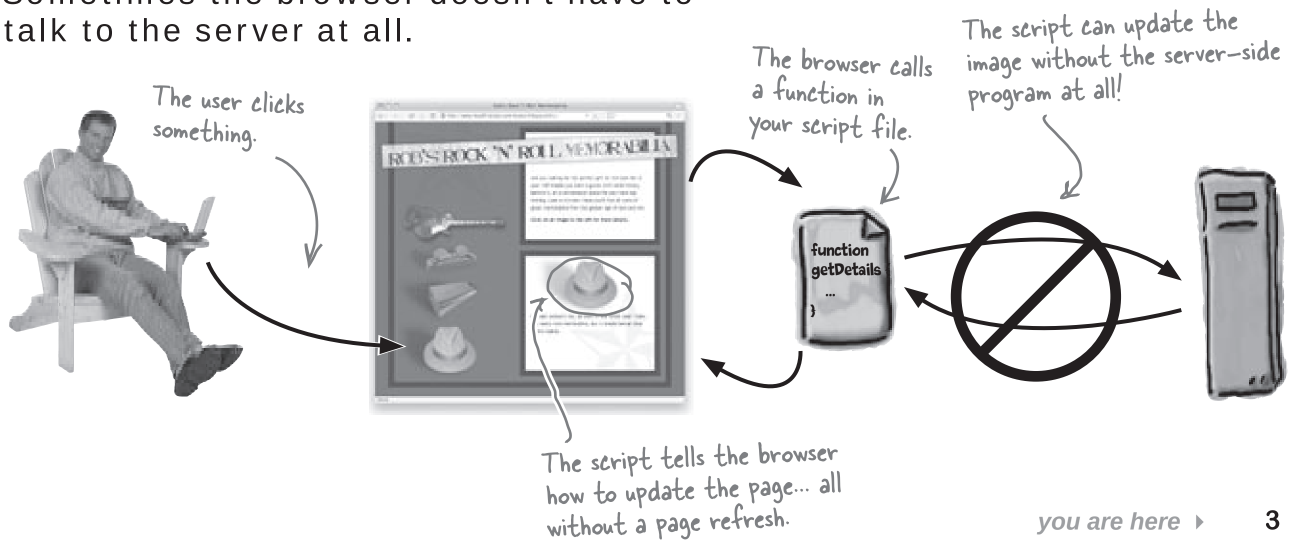
Web pages reinvented

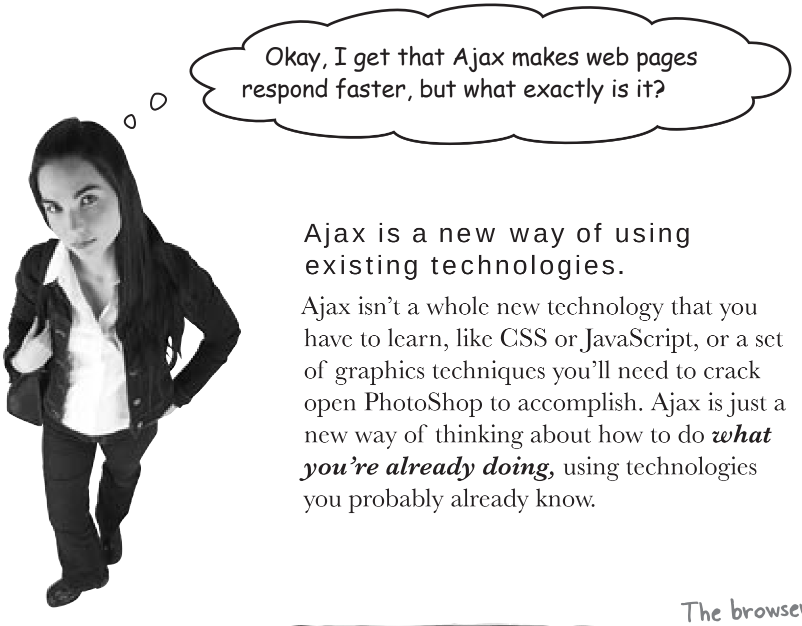
Using Ajax, your pages and applications only ask the server for what they really need—just the parts of a page that need to change, and just the parts that the server has to provide. That means less traffic, smaller updates, and **less time sitting around** waiting for page refreshes.

With Ajax, the browser only sends and receives the parts of a page that need to change.



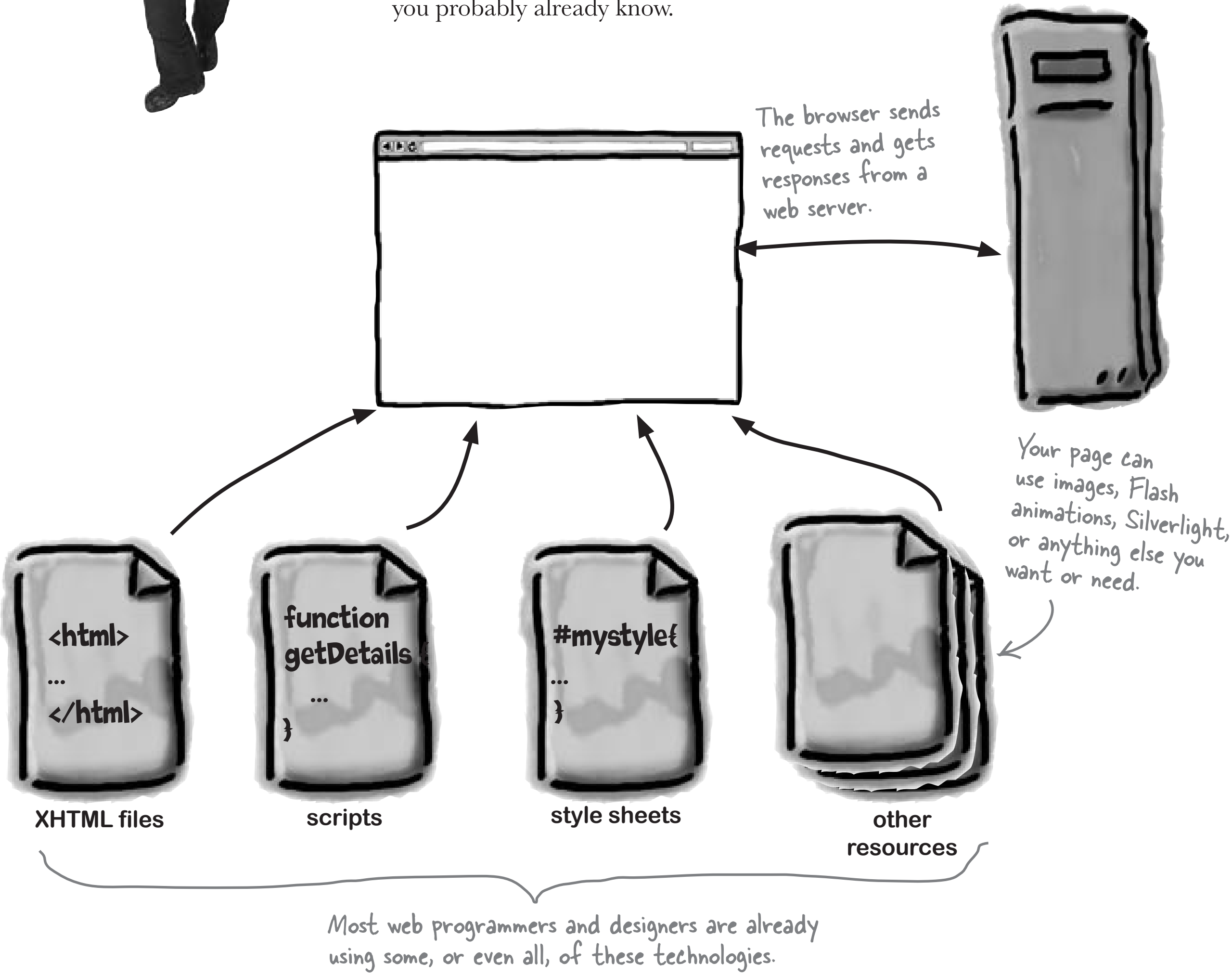
Sometimes the browser doesn't have to talk to the server at all.





Ajax is a new way of using existing technologies.

Ajax isn't a whole new technology that you have to learn, like CSS or JavaScript, or a set of graphics techniques you'll need to crack open PhotoShop to accomplish. Ajax is just a new way of thinking about how to do ***what you're already doing,*** using technologies you probably already know.



So what makes a page "Ajax"?

Ajax is a way of designing and building web pages that are as interactive and responsive as desktop applications. So what does that mean for you? You handle things at the client's browser whenever you can. Your pages make **asynchronous requests** that allow the user to keep working instead of waiting for a response. You only update the things on your pages that actually change. And best of all, an Ajax page is built using standard Internet technologies, things you probably already know how to use, like:

- XHTML
- Cascading Style Sheets
- JavaScript

Ajax applications also use a few things that have been around for a while but may be new to you, like:

- The XmlHttpRequest
- XML & JSON
- The DOM

We'll look at all of these in detail before we're through.

An asynchronous request is a request that occurs behind the scenes.

Your users can keep working while the request is taking place.

there are no Dumb Questions

Q: Doesn't Ajax stand for "Asynchronous JavaScript and XML"?

A: Sort of. Since lots of pages that are considered "Ajax" don't use JavaScript or XML, it's more useful to define Ajax as a way of building web pages that are as responsive and interactive as desktop applications, and not worry too much about the exact technologies involved.

Q: What exactly does "asynchronous" mean?

A: In Ajax, you can make requests to the server without making your user wait around for a response. That's called an asynchronous request, and it's the core of what Ajax is all about.

Q: But aren't all web pages asynchronous? Like when a browser loads an image while I'm already looking at the page?

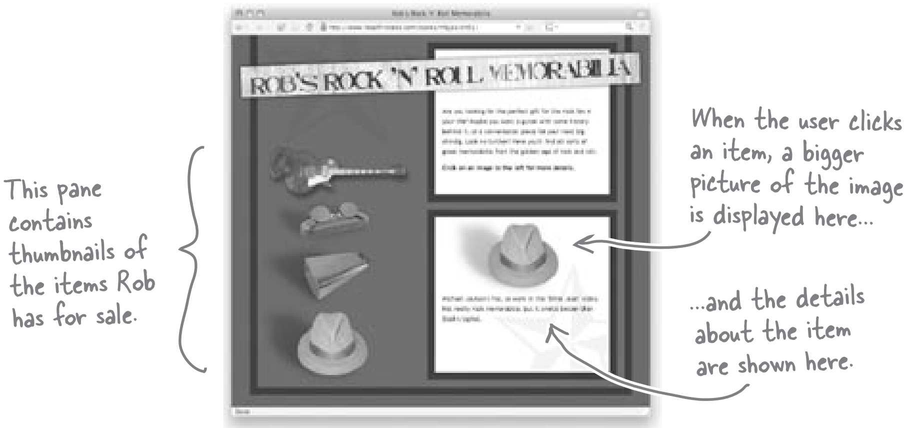
A: *Browsers* are asynchronous, but the standard web page isn't. Usually when a web page needs information from a server-side program, everything comes to a complete stop until the server responds... unless the page makes an asynchronous request. And that' what Ajax is all about.

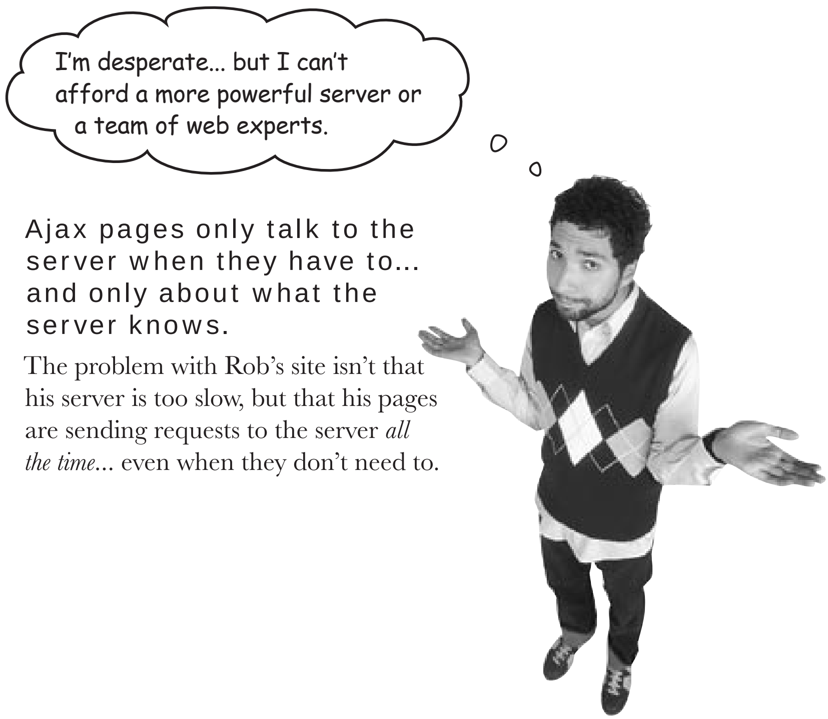
Q: But all Ajax pages use that XMLHttpRequest object, right?

A: Nope. Lots do, and we'll spend a couple of chapters mastering `XMLHttpRequest`, but it's not a requirement. In fact, lots of apps that are considered Ajax are more about user interactivity and design than any particular coding technique.

Rob's Rock 'n' Roll Memorabilia

Meet Rob. He's put all his savings into an online rock n' roll memorabilia store. The site looks great, but he's still been getting tons of complaints. Customers are clicking on the thumbnail images on the inventory page, but the customers' browsers are taking forever before they show information about the selected item. Some of Rob's users are hanging around, but most have just stopped coming to Rob's online shop altogether.



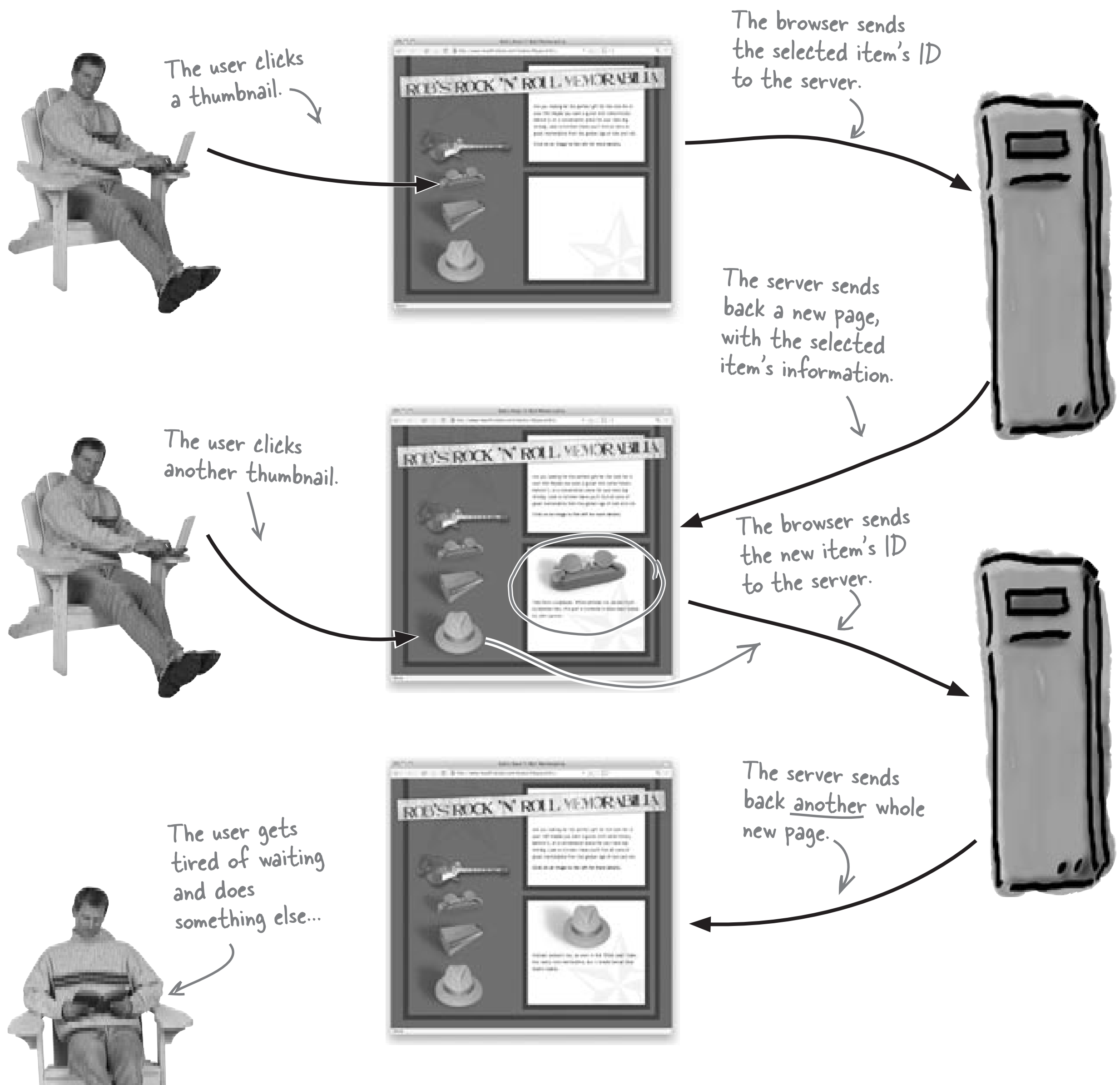


Ajax pages only talk to the server when they have to... and only about what the server knows.

The problem with Rob's site isn't that his server is too slow, but that his pages are sending requests to the server *all the time*... even when they don't need to.

Sharpen your pencil

Here's what Rob's online store does right now. What's wrong with this picture?



How would Ajax change this diagram? Write down what you think ***should*** happen on Rob's site.

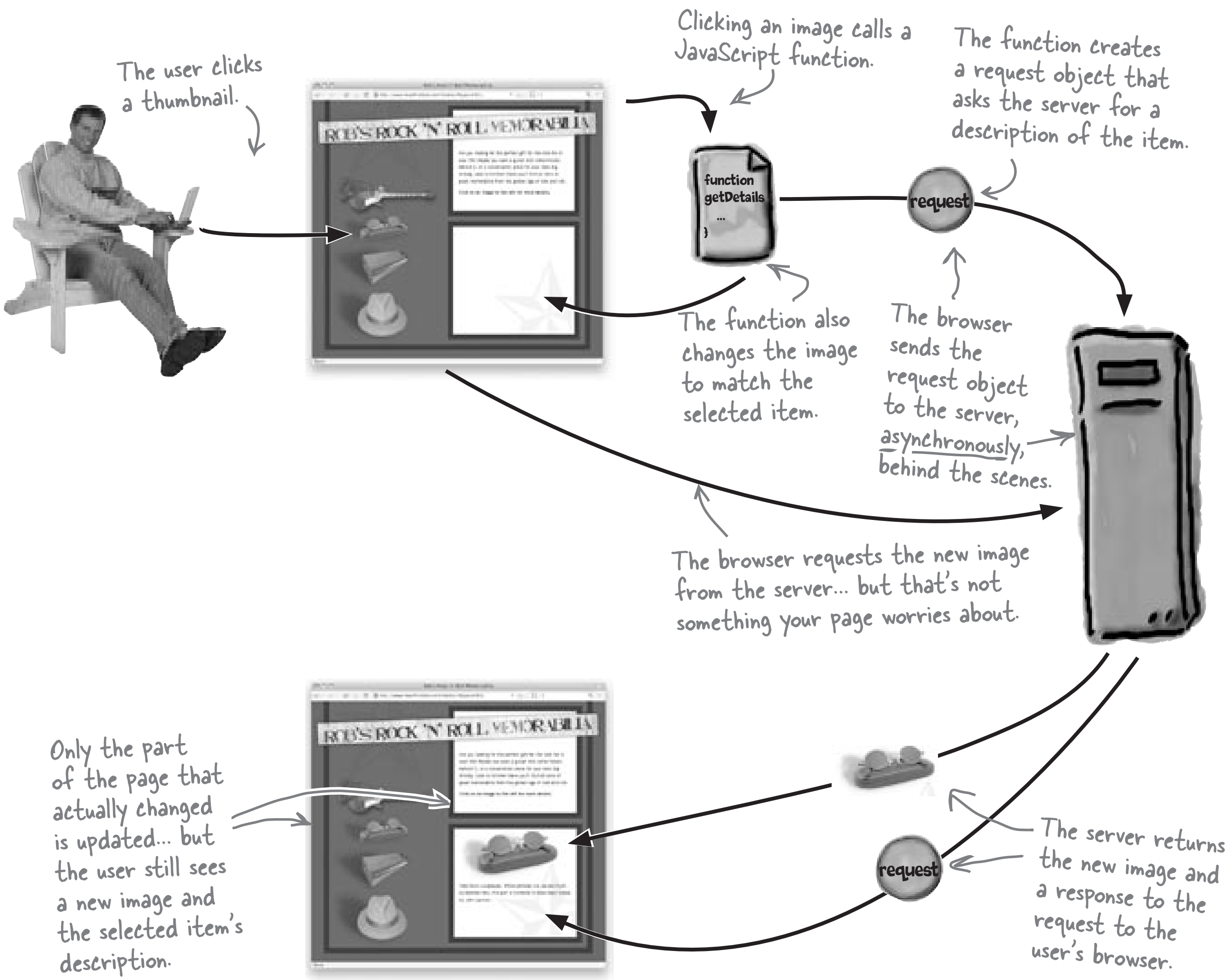
....

....

....

Sharpen your pencil Solution

Your job was to think about how Ajax could help save Rob's site... and his business. With Ajax, we can **completely remove** all the page refreshes on his inventory page. Here's what that would look like:



BULLET POINTS

- Asynchronous requests allow more than one thing to happen at the same time.
- Only the part of a web page that needs to change gets updated.
- The page isn't frozen while the server is returning data to the browser.

Sharpen your pencil

Put a checkmark next to the benefits that you think Ajax can provide to your web applications.

- [ ] **The browser can request multiple things from the server at the same time.**
- [ ] **Browser requests return a lot faster.**
- [ ] **Colors are rendered more faithfully.**
- [ ] **Only the parts of the page that actually change are updated.**
- [ ] **Server traffic is reduced.**
- [ ] **Pages are less vulnerable to compatibility issues.**
- [ ] **The user can keep working while the page updates.**
- [ ] **Some changes can be handled without a server round-trip.**
- [ ] **Your boss will love you.**
- [ ] **Only the parts of the page that actually change are updated.**

BRAIN BARBELL

Not all pages will reap every benefit of Ajax. In fact, some pages wouldn't benefit from Ajax at all. Which of the benefits that you checked off above do you think Rob's page will see?

Sharpen your pencil Solution

Remember, not every page is going to see all these benefits...

With asynchronous requests, you can make sure the browser works behind the scenes, and avoid interrupting your users with full-page refreshes.

- [x] **The browser can request multiple things from the server at the same time.**

This is only true sometimes. The speed of a request and response depends on what the server is returning. And it's possible to build Ajax pages that are slower than traditional pages.

- [x] **Browser requests return a lot faster.**
- [ ] **Colors are rendered more faithfully.**

Color rendering is dictated by the user's monitor, not your app.

- [x] **Only the parts of the page that actually change are updated.**

It's possible to make smaller, more focused requests with Ajax. Be careful, though... it's also easy to make a lot more requests—and increase traffic—because you can make all of those requests asynchronously.

- [x] **Server traffic is reduced.**

Because Ajax pages rely on technologies in addition to XHTML, compatibility issues can actually be a bigger problem with Ajax. Test, test, test your apps on the browsers your users have installed.

- [ ] **Pages are less vulnerable to compatibility issues.**

Sometimes you want a user to wait on the server's response, but that doesn't mean you can't still use Ajax. We'll look at synchronous vs. asynchronous requests more in Chapter 5.

- [x] **The user can keep working while the page updates.**

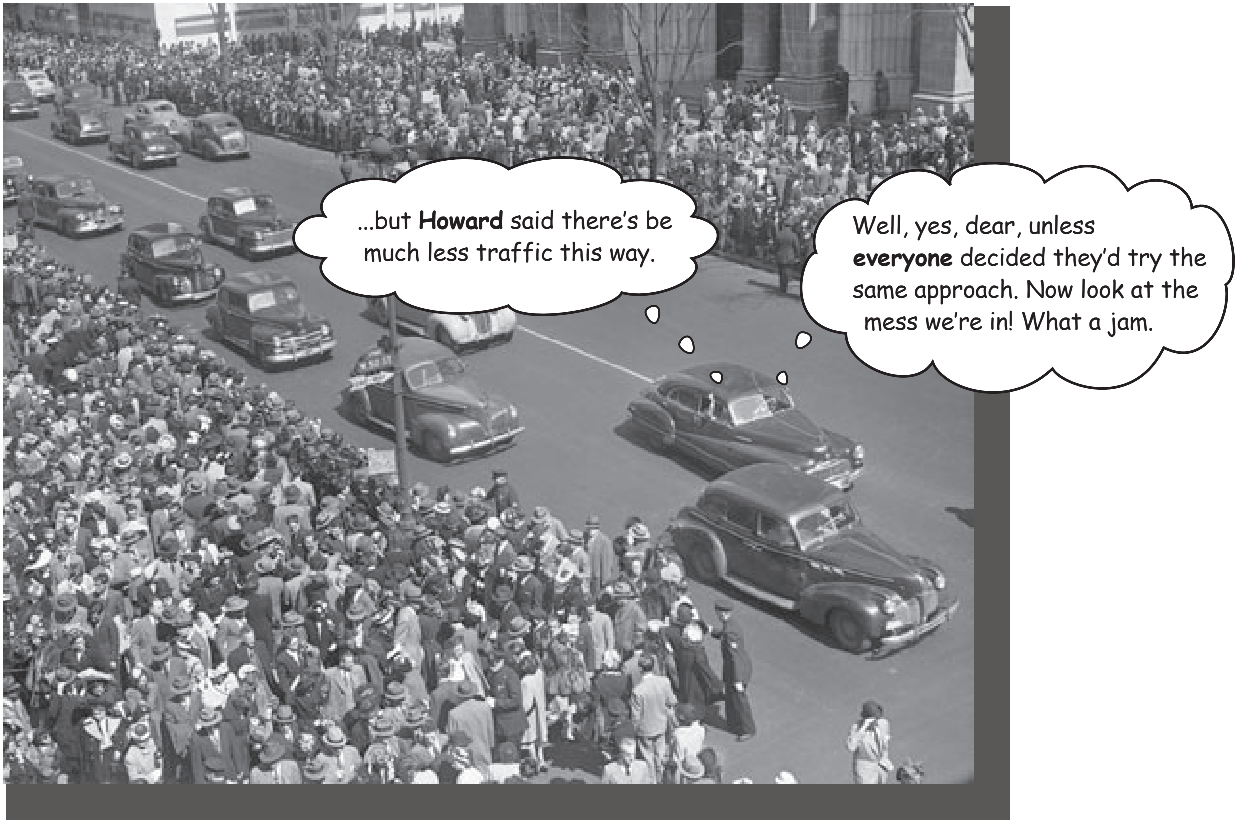
Handling things at the browser can make your web application feel more like a desktop application.

- [x] **Some changes can be handled without a server round-trip.**
- [?] **Your boss will love you.**

If you use Ajax in a way that helps your apps, the boss will love you. But you shouldn't use Ajax everywhere... more on that later.

- [x] **Only the parts of the page that actually change are updated.**

Yes, this is the second time this shows up in the list. It's that important!



there are no Dumb Questions

Q: First you said Ajax was the web reinvented. Now it's increasing server traffic. Which is it?

A: Sometimes it's both! Ajax is one way to make requests, get responses, and build responsive web apps. But you've still got to be smart when deciding whether an asynchronous request or a regular synchronous request would be a better idea.

Q: How do I know when to use Ajax and asynchronous requests, and when not to?

A: Think about it like this: if you want something to go on while your user's still working, you probably want an asynchronous request. But if your user needs information or a response from your app before they continue, then you want to make them wait. That usually means a synchronous request.

Q: So for Rob's online store, since we want users to keep browsing while we're loading product images and descriptions, we'd want an asynchronous request. Right?

A: Exactly. That particular part of Rob's app—checking out different items—shouldn't require the user to wait every time they select a new item. So that's a great place to use Ajax and make an asynchronous request.

Q: And how do I do that?

A: Good question. Turn the page, and let's get down to actually using Ajax to fix up Rob's online store.

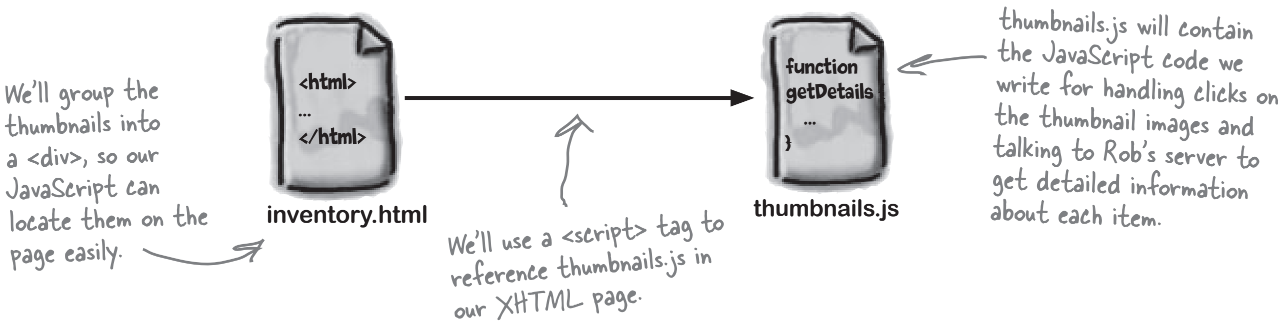
Ajax and rock 'n' roll in 5 steps

Let's use Ajax to fix up Rob's online store, and get his impatient customers back. We'll need to make some changes to the existing XHTML page, code some JavaScript, and then reference the script in our XHTML. When we're done, the page won't need to reload at all, and only the things that need to change will get updated when users click on the thumbnail images.

Here's what we're going to do:

1 Modify the XHTML web page

We need to include the JavaScript file we're going to write and add some `div`s and `id`s, so our JavaScript can find and work with different parts of the web page.



2 Write a function to initialize the page

When the inventory page first loads, we'll need to run some JavaScript to set up the images, get a request object ready, and make sure the page is ready to use.

This tells the browser to run the initPage() function as soon as the page loads up.

```
window.onload = initPage;
function initPage() {

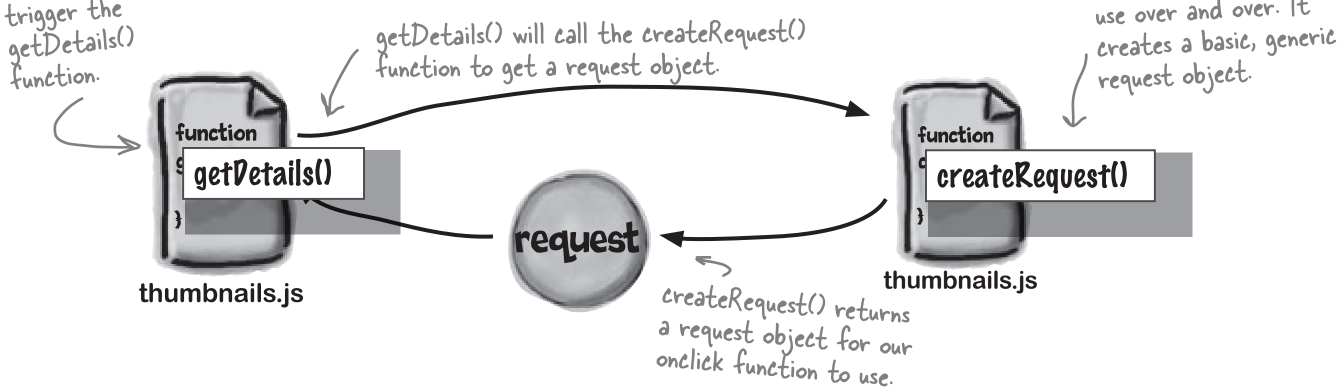
  // setup the images
  // create a request object
}
```

We'll write code in initPage() to initialize all the thumbnail images, and set up onClick event handlers for each image.



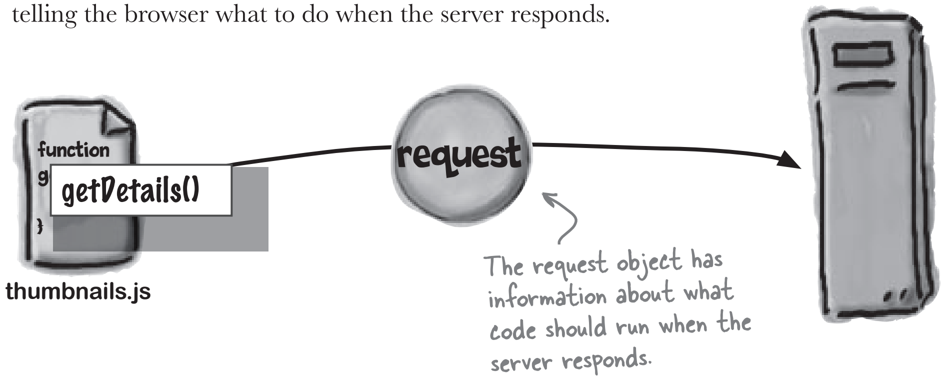
3 Write a function to create a request object

We need a way to talk to the server and get details about each piece of memorabilia in Rob's inventory. We'll write a function to create a request object to let our code talk to the server; let's call it `createRequest()`. We can use that function whenever a thumbnail is clicked to get a new request started.



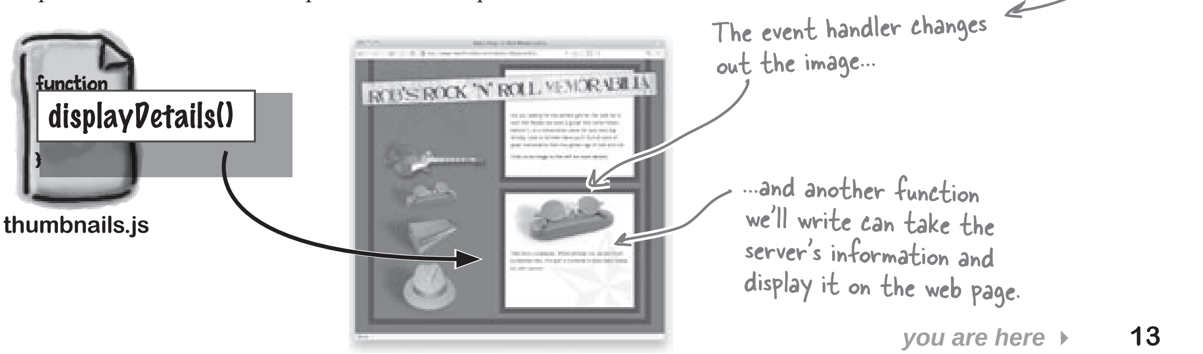
4 Get an item's details from the server

We'll send a request to Rob's server in `getDetails()`, telling the browser what to do when the server responds.

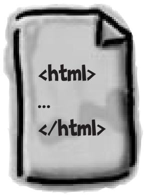


5 Display the item's details

We can change the image to display in `getDetails()`. Then, we need another function, `displayDetails()`, to update the item's description when the server responds to our requests.



Step 1: Modify the XHTML



inventory.html

Let's start with the easy part, the XHTML and CSS that create the page. Here's Rob's current version of the inventory page with a few additions we'll need:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Rob's Rock 'n' Roll Memorabilia</title>
  <link rel="stylesheet" href="css/default.css" />
  <script src="scripts/thumbnails.js" type="text/javascript"></script>
</head>
<body>
  <div id="wrapper">
    <img src="images/logotypeLeft.png" alt="Rob's Rock 'n' Roll Memorabilia"
         width="394" height="91" id="logotypeLeft" />
    <img src="images/logotypeRight.png" alt="Rob's Rock 'n' Roll Memorabilia"
         width="415" height="92" id="logotypeRight" />
    <div id="introPane">
      <p>Are you looking for the perfect gift for the rock fan in your life?
         Maybe you want a guitar with some history behind it, or a conversation
         piece for your next big shindig. Look no further! Here you'll find all
         sorts of great memorabilia from the golden age of rock and roll.</p>
      <p><strong>Click on an image to the left for more details.</strong></p>
    </div>
    <div id="thumbnailPane">
      <img src="images/itemGuitar.jpg" width="301" height="105" alt="guitar"
           title="itemGuitar" id="itemGuitar" />
      <img src="images/itemShades.jpg" alt="sunglasses" width="301" height="88"
           title="itemShades" id="itemShades" />
      <img src="images/itemCowbell.jpg" alt="cowbell" width="301" height="126"
           title="itemCowbell" id="itemCowbell" />
      <img src="images/itemHat.jpg" alt="hat" width="300" height="152"
           title="itemHat" id="itemHat" />
    </div>
    <div id="detailsPane">
      <img src="images/blank-detail.jpg" width="346" height="153" id="itemDetail" />
      <div id="description"></div>
    </div>
  </div>
</body>
</html>
```

You need to add a reference to thumbnails.js. That's the script we'll be writing in this chapter.

This <div> holds the small, clickable images.

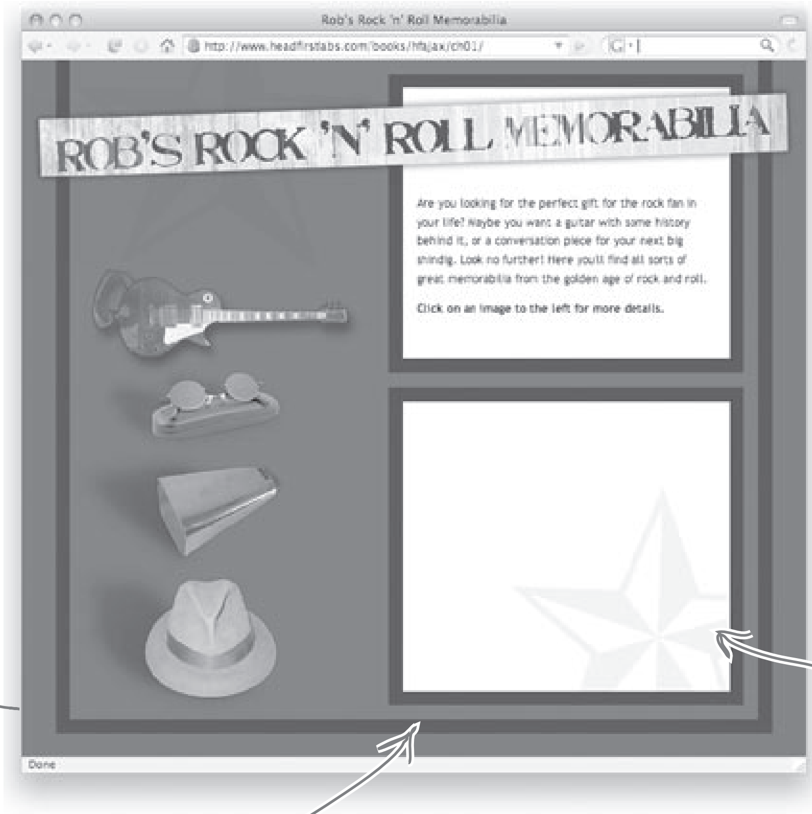
This <div> is where details about each item should go.

We'll put item details in here with our JavaScript.



It's time to get the samples and get going.

Download the examples for the book at **www.headfirstlabs.com**, and find the **chapter01** folder. Now open the **inventory.html** file in a text editor, and make the changes shown above.



To Do

- [x] **Modify the XHTML**
- [ ] Initialize the page
- [ ] Create a request object
- [ ] Get the item's details
- [ ] Display the details

Here's a short version of the steps from pages 12 and 13 that we can use to work through Rob's page.

Start out with no item detail and a blank area for the item's description to go in when something's selected.

This is the cascading style sheet for Rob's page. We'll use the id values on the <div> elements to style the page, and also later in our JavaScript code.

```
body {
  background: #333;
  font-family: Trebuchet MS, Verdana, Helvetica, Arial, san-serif;
  margin: 0;
  text-align: center;
}

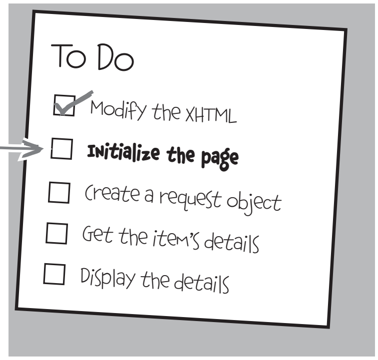
p { font-size: 12px; line-height: 20px; }
a img { border: 0; }

#wrapper {
  background: #750505 url('../images/bgWrapper.png') 8px 0 no-repeat;
  border: solid #300;
  border-width: 0 15px 15px 15px;
  height: 700px;
  margin: 0 auto;
  ...etc...
```

There's a lot more CSS... you can see the complete file by downloading the examples from the Head First Labs web site.

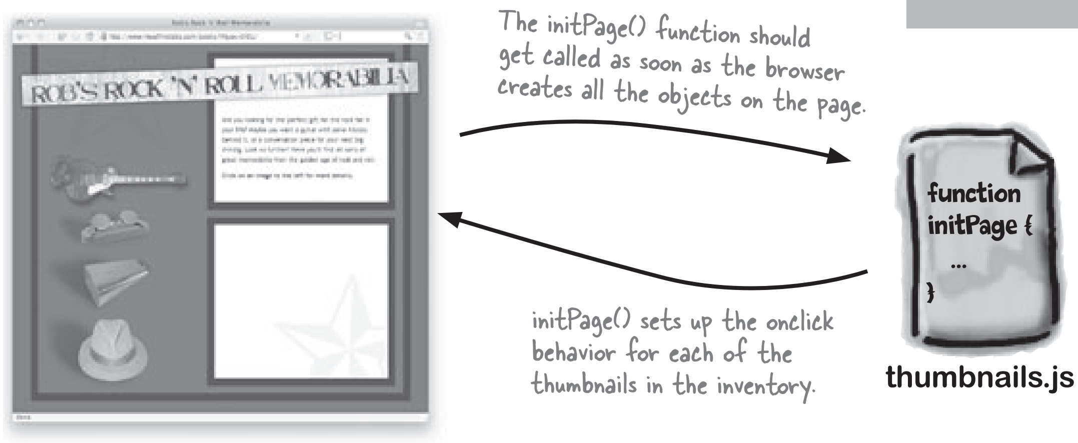


rocknroll.css



Step 2: Initialize the JavaScript

We need to create the `thumbnails.js` file, and add a JavaScript function to set up the initial event handlers for each thumbnail image in the inventory. Let's call that function `initPage()`, and set it to run as soon as the user's window loads the inventory page.



To set up the onclick behavior for the thumbnails, the initPage() function has to do two things:

1. **Find the thumbnails on the page**
   The thumbnails are contained in a `div` called "thumbnailPane," so we can find that `div,` and then find each image within it.

2. **Build the onclick event handler for each thumbnail**
   Each item's full-size image is named with the title of the thumbnail image plus "-detail". For example, the detail image for the thumbnail with the title `FenderGuitar` is `FenderGuitar-detail.png`. That lets us work out the name of the image in our JavaScript.

   The event handler for each thumbnail should set the `src` tag for the detail image (the one with an id of "itemDetail") to the detail image (for example, `FenderGuitar-detail.png`). Once you've done that, the browser will automatically display the new image using the name you supplied.



Code Magnets

The code for the initPage function is all scrambled up on the fridge. Can you put back the pieces that fell off? Remember to set an event handler to run the initPage() function when the user's window loads, too.

```
// create the onclick function
image.onclick = function() {
}
}
}
// find the thumbnails on the page
// find the full-size image name
function initPage() {
image = thumbs[i];
for (var i=0; i<thumbs.length; i++) {
window.onload = initPage;
document.getElementById("itemDetail").src = detailURL;
getDetails(this.title);
thumbs = document.getElementById("thumbnailPane").getElementsByTagName("IMG");
// set the handler for each image
detailURL = 'images/' + this.title + '-detail.jpg';
```

In an event handler, like onclick, you can get a reference to the object the event occurred on with the "this" keyword.

Code Magnet Solution

```
window.onload = initPage;

function initPage() {
  // find the thumbnails on the page
  thumbs = document.getElementById("thumbnailPane").getElementsByTagName("img");

  // set the handler for each image
  for (var i=0; i<thumbs.length; i++) {
    image = thumbs[i];

    // create the onclick function
    image.onclick = function() {
      // find the full-size image name
      detailURL = 'images/' + this.title + '-detail.jpg';
      document.getElementById("itemDetail").src = detailURL;
      getDetails(this.title);
    }
  }
}
```

This sets initPage() up to run once the user's browser loads the page.

All these "get..." functions use the DOM to look up something on the XHTML page.

Don't worry too much about this now... we'll talk about the DOM in detail a bit later.

These are the same ids we used in the CSS to style the page.

We want to do this once for every thumbnail.

JavaScript lets you define functions without giving them an explicit name.

When an image is clicked, that image's title is used to figure out the detail image's URL.

This function is run whenever a thumbnail image is clicked.

Clicking on a thumbnail changes the detail image's src attribute, and then the browser displays the new image.

Don't forget all the closing brackets, or your JavaScript won't run.

Test Drive

Create thumbnails.js, add the initPage() function, and give the inventory page a whirl.

Create a file named **`thumbnails.js`** in a text editor. Add the code shown on page 18, and then load `inventory.html` in your browser. `initPage()` should run when the page loads, and you're ready to try out the detail images...



Click here....

...and an image is displayed here.



The item's details won't show up yet, but the right image should appear.

You can check another item off the To Do list for Rob's inventory page.

To Do

- [x] Update the XHTML
- [x] **Initialize the JavaScript**
- [ ] Create a request object
- [ ] Get the item's details
- [ ] Display the details

Step 3: Create a request object

When users click on an item's image, we also need to send a request to the server asking for that item's detailed information. But before we can send a request, we need to create the request object.

The bad news is that this is a bit tricky because different browsers create request objects in different ways. The good news is that we can create a function that handles all the browser-specific bits.

Go ahead and create a new function in `thumbnails.js` called `createRequest()`, and add this code:

To Do

- [x] Modify the XHTML
- [x] Initialize the page
- [ ] **Create a request object**
- [ ] Get the item's details
- [ ] Display the details



Ready Bake Code

Ready Bake code is code that you can just type in and use... but don't worry, you'll understand all of this in just another chapter or two.

```
function createRequest() {
  try {

    request = new XMLHttpRequest();

  } catch (tryMS) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");

    } catch (otherMS) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = null;
      }
    }
  }

  return request;
}
```

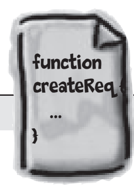
This line tries to create a new request object, but it won't work for every browser type.

The first approach failed, so try again using a different type of object.

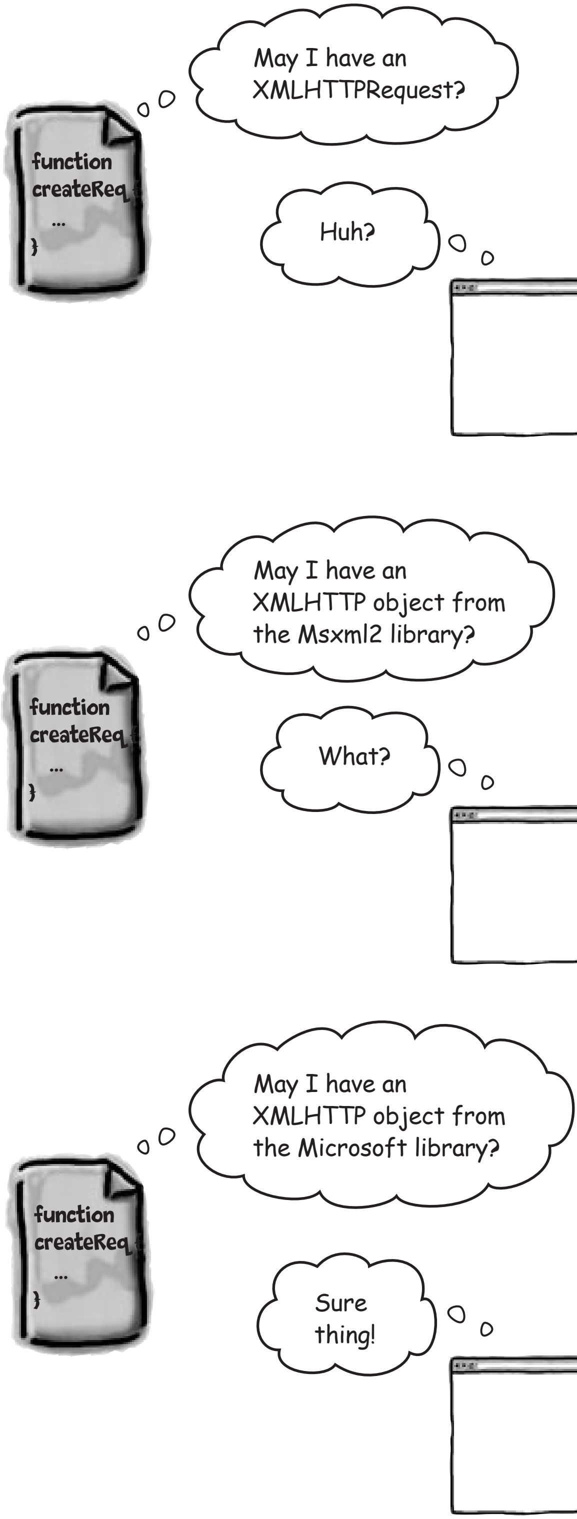
That didn't work either, so try one more thing.

If the code gets here, nothing worked. Return a null so that the calling code will know there was a problem.

This either returns a request object, or "null" if nothing worked.



thumbnails.js



there are no Dumb Questions

Q: Am I supposed to understand all of this?

A: No, you're not. For now, just try to get a general idea of how all this looks and the way the pieces fit together. Focus on the big picture, and then we'll start to fill in the gaps in later chapters.

Q: So what's an XMLHttpRequest?

A: `XMLHttpRequest` is what most browsers call the request object that you can send to the server and get responses from without reloading an entire page.

Q: Well, if that's an XMLHttpRequest, what's an ActiveXObject?

A: An `ActiveXObject` is a Microsoft-specific programming object. There are two different versions, and different browsers support each. That's why there are two different code blocks, each trying to create a different version of `ActiveXObject`.

Q: And the request object is called XMLHTTP in a Microsoft browser?

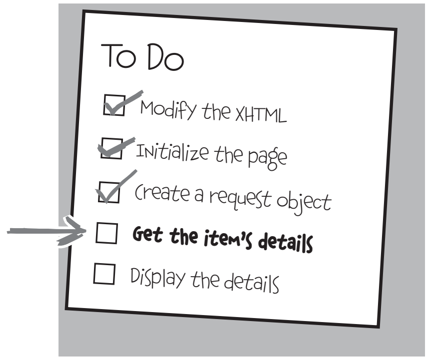
A: That's the *type* of the object, but you can call your variable anything you'd like; we've been using `request`. Once you have the `createRequest()` function working, you never have to worry about these different types again. Just call `createRequest()`, and then assign the returned value to a variable.

Q: So my users don't need to be using a specific browser?

A: Right. As long as their browsers have JavaScript enabled, your users can be running any browser they want.

Q: What if they don't have JavaScript enabled?

A: Unfortunately, Ajax applications require JavaScript to run. So users who have JavaScript disabled aren't going to be able to use your Ajax applications.The good news is that JavaScript is usually enabled by default, so anyone who has disabled JavaScript probably knows what they're doing, and could turn JavaScript support back on if they wanted to use your Ajax app.



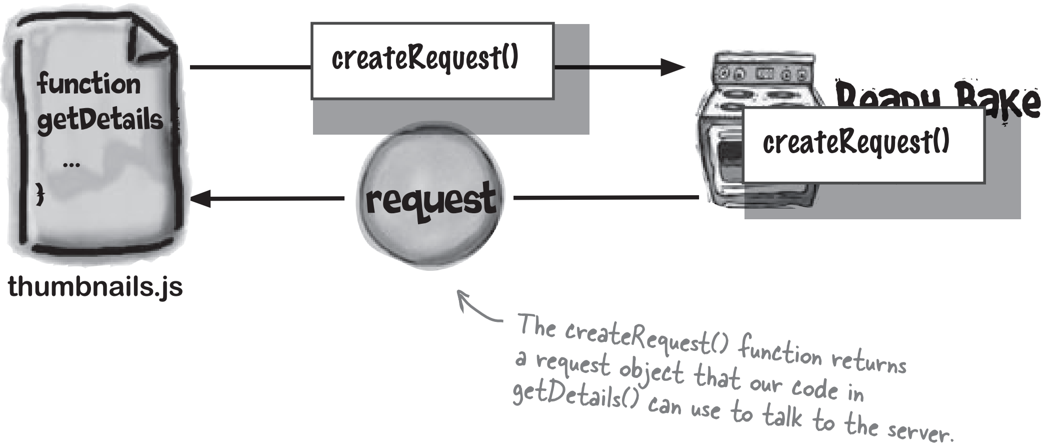
Step 4: Get the item's details

Once a user clicks on an item in the inventory, we need to send a request to the server and ask for the description and details for that item. We've got a request object, so here is where we can use that.

And it turns out that no matter what data you need from the server, the basic process for making an Ajax request always follows the same pattern:

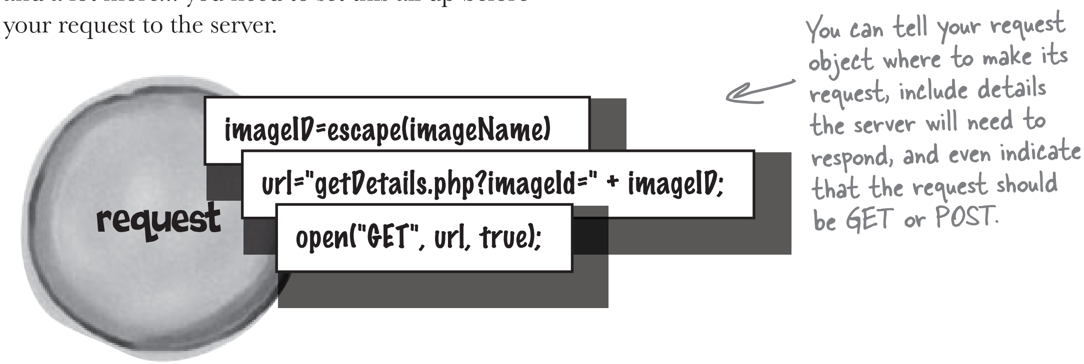
1. **Get a request object**

   We've already done the work here. We just need to call `createRequest()` to get an instance of the request object and assign it to a variable.



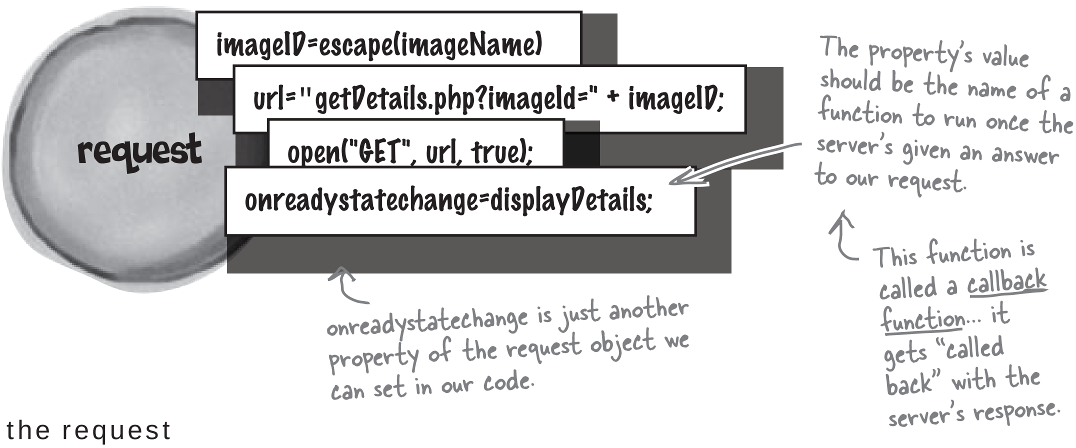
2. **Configure the request object's properties**

   The request object has several properties you'll need to set. You can tell it what URL to connect to, whether to use GET or POST, and a lot more... you need to set this all up before you make your request to the server.



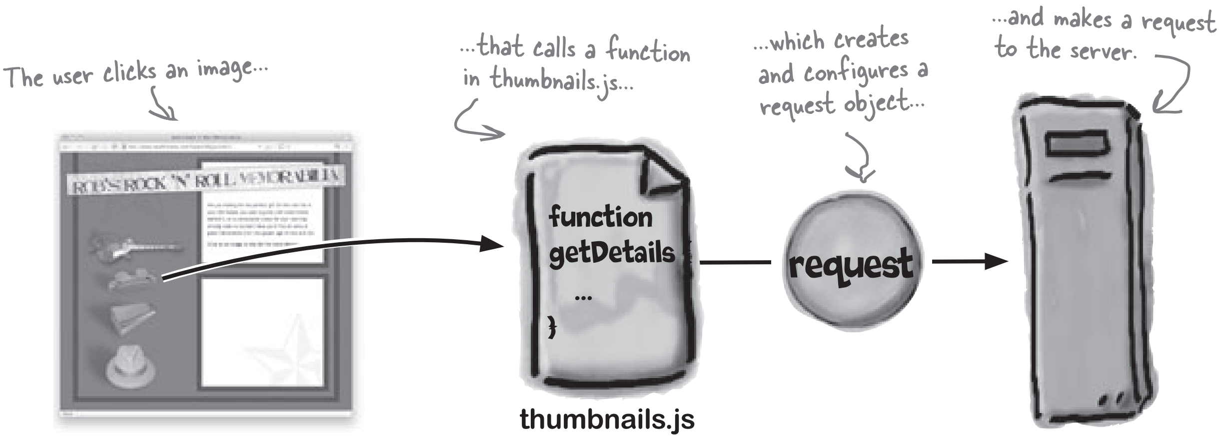
3 Tell the request object what to do when the server responds

So what happens when the server responds? The browser looks at another property of the request object, called `onreadystatechange`. This lets us assign a **callback function** that should run when the server responds to our request.



4 Make the request

Now we're ready to send the request off to the server and get a response.



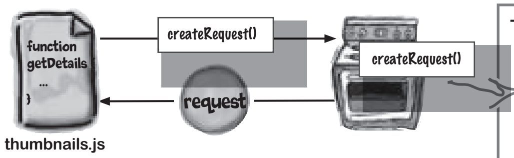
BRAIN POWER

Why do you think the callback function is assigned to a property called onreadystatechange? What do you think that property name means?

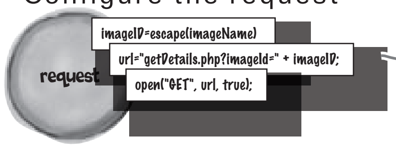
Let's write the code for requesting an item's details

Once we know what our function needs to do, it's pretty easy to write the code. Here's how the steps map to actual JavaScript in `thumbnails.js`:

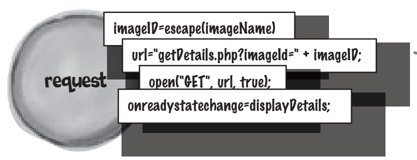
1. Get a request object



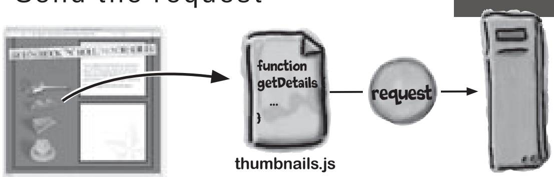
2. Configure the request



3. Set the callback function



4. Send the request



The onclick handler for each inventory image calls this function and passes in the clicked img element's title attribute, which is the name of the item the image represents.

```
function getDetails(itemName) {
  request = createRequest();
  if (request==null) {
    alert("Unable to create request");
    return;
  }
  var url= "getDetails.php?ImageID=" +
    escape(itemName);
  request.open("GET",url,true);
  request.onreadystatechange = displayDetails;
  request.send(null);
}
```

We've got to check to make sure the request object isn't null... that's how we know if there was a problem creating the object.

escape() takes care of any characters that might be a problem in a request URL string.

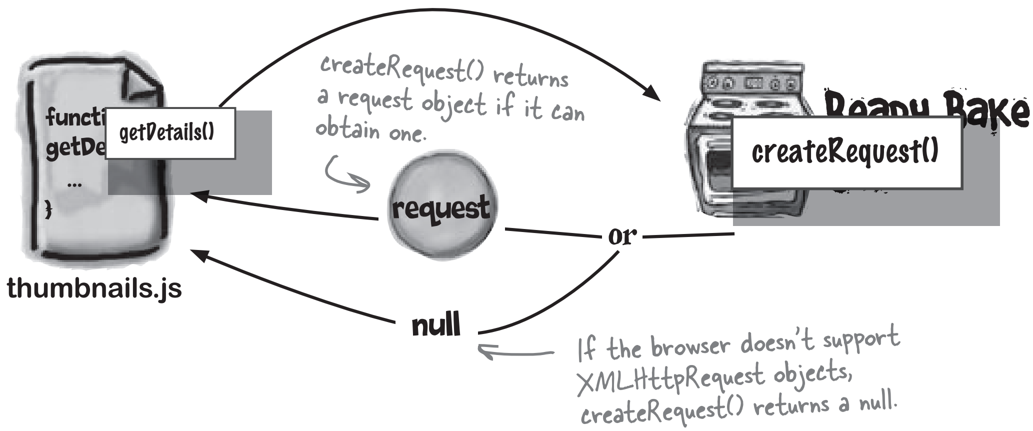




Add the `getDetails()` function to your version of `thumbnails.js`

Always make sure you have a request object before working with it

The first thing `getDetails()` does is call `createRequest()` to get a request object. But you've still got to make sure that object was actually created, even though the details of that creation are abstracted away in the `createRequest()` function:



And here's how that looks in our code...

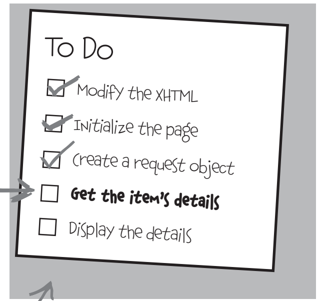
This line asks for an instance of the request object and assigns it to the variable "request."

```
function getDetails(itemName) {
  request = createRequest();
  if (request==null) {
    alert("Unable to create request");
    return;
  }
  var url= "getDetails.php?ImageID=" +
    escape(itemName);
  request.open("GET",url,true);
  request.onreadystatechange = displayDetails;
  request.send(null);
}
```

createRequest() returns a null if it can't get a request object. So if we wind up in this bit of code, we know something's gone wrong. We'll display an error to the user and exit the function.



The request object is just an object



We're still working on getting the details for an item.

A request object is just a "normal" JavaScript object, and that means you can set properties on it and call methods. We can talk to the server by putting information in the request object.

```
function getDetails(itemName) {
  request = createRequest();
  if (request==null) {
    alert("Unable to create request");
    return;
  }
  var url= "getDetails.php?ImageID=" +
    escape(itemName);
  request.open("GET",url,true);
  request.onreadystatechange = displayDetails;
  request.send(null);
}
```

This line tells the request object the URL to call. We send along the name of the item, so the server knows which details to send.

These parameters tell the request object how we want it to connect to the server.



thumbnails.js

Let's break open() down a bit...

`request.open(` — The open() method initializes the connection.

`"GET"` — "GET" indicates how to send the data (the other option is "POST").

`url` — This is the url for the server-side script that will respond to the request.

`true);` — This means that the request should be asynchronous. That is, the code in the browser should continue to execute while it's waiting for the server to respond.

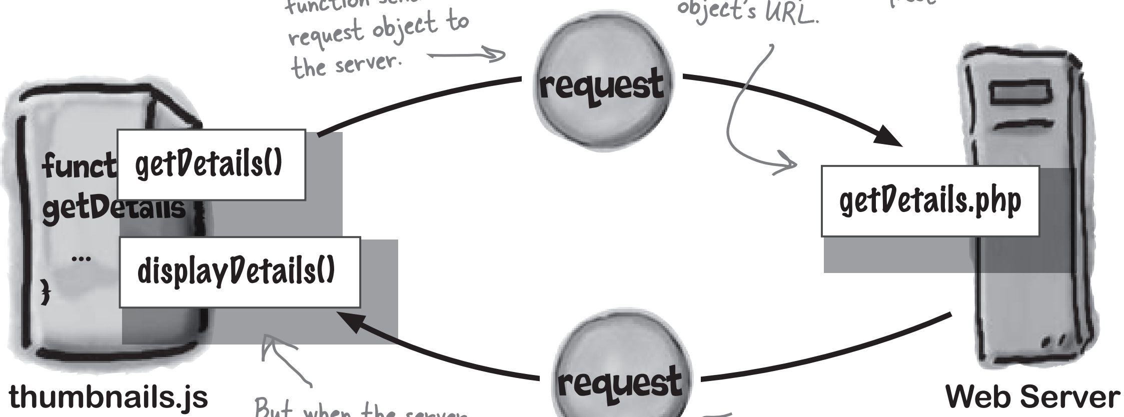
there are no Dumb Questions

Q: Are there other properties of the request object?

A: Sure. You've already seen `onreadystatechange`, and when you need to send XML or more complicated data to the server, then there are several others you'll use. For now, though, we just need the `open()` method and `onreadystatechange`.

Hey, server... will you call me back at displayDetails(), please?

The properties of the request object tell the server what to do when it receives the request. One of the most important is the `onreadystatechange` property, which we're setting to the name of a function. This function, referred to as a **callback**, tells the browser what code to call when the server sends back information.




```
function getDetails(itemName) {
  request = createRequest();
  if (request==null) {
    alert("Unable to create request");
    return;
  }
  var url= "getDetails.php?ImageID=" +
    escape(itemName);
  request.open("GET",url,true);
  request.onreadystatechange = displayDetails;
  request.send(null);
}
```



This is the line that tells the browser what code to call when the server responds to the request.

This is a reference to a function, not a function call. So make sure you don't include any parentheses at the end of the function name.

Use send() to send your request

All that's left to do is actually send the request, and that's easy... just use the `send()` method on the request object.

```
function getDetails(itemName) {
  request = createRequest();
  if (request==null) {
    alert("Unable to create request");
    return;
  }
  var url= "getDetails.php?ImageID=" +
    escape(itemName);
  request.open("GET",url,true);
  request.onreadystatechange = displayDetails;
  request.send(null);
}
```

You're sending the request here...

...and this means you're not sending any extra data with the request.



thumbnails.js



Aren't you forgetting something? We don't want to send null; we want to send the item name.

You can send data in your URL string.

The request object allows us to send all kinds of data in a variety of ways. In `getDetails()`, the item name is part of the URL string:

```
var url= "getDetails.php?ImageID=" +
  escape(itemName);
```

Since that's part of the URL sent to the server, we don't need to send anything else to the server in the `send()` method. Instead, we just pass `null`... which means "nothing."

Asynchronous apps make requests using a JavaScript object, not a form submit.

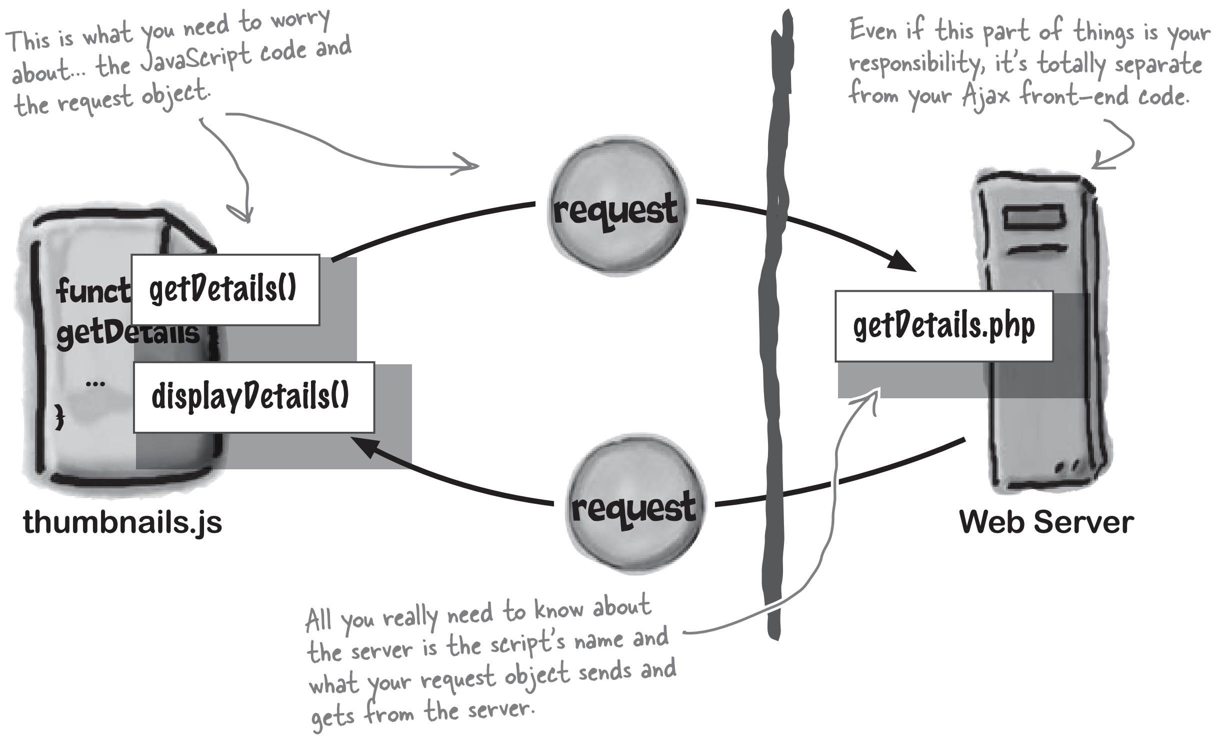


So what about the server-side code?

The server-side code is... ...on the server.

That sounds obvious, but lots of times, you don't have to (or even get to) write the code your web application is talking to. Instead, you work with an existing program, where you know the inputs and outputs, or tell another group what you need.

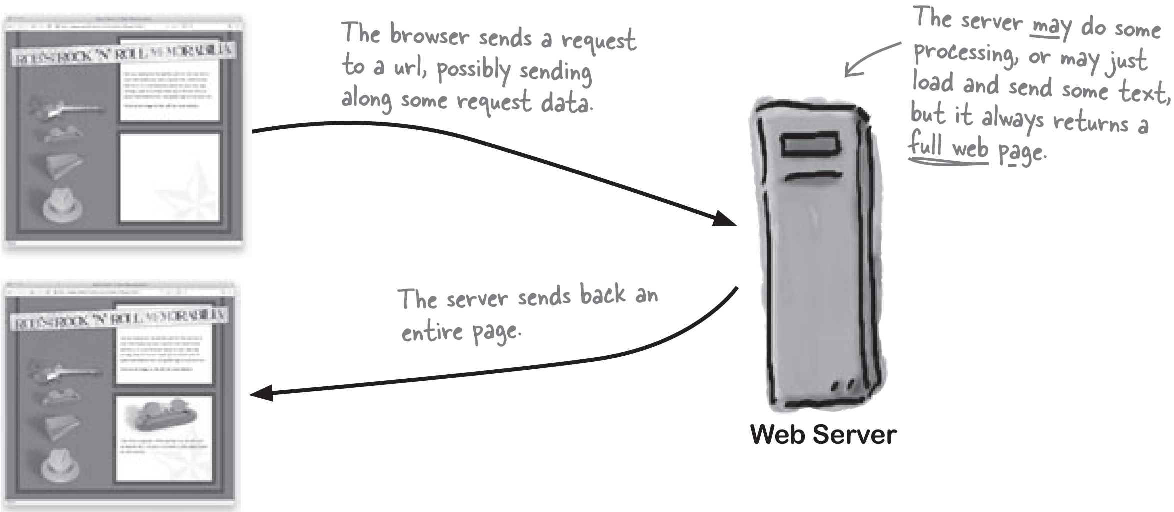
Not only that, but you might also have one server-side program that's written in PHP, and another in ASP.NET... and other than the URL, you don't have to change your JavaScript code at all. Take a look:



The server usually returns data to Ajax requests

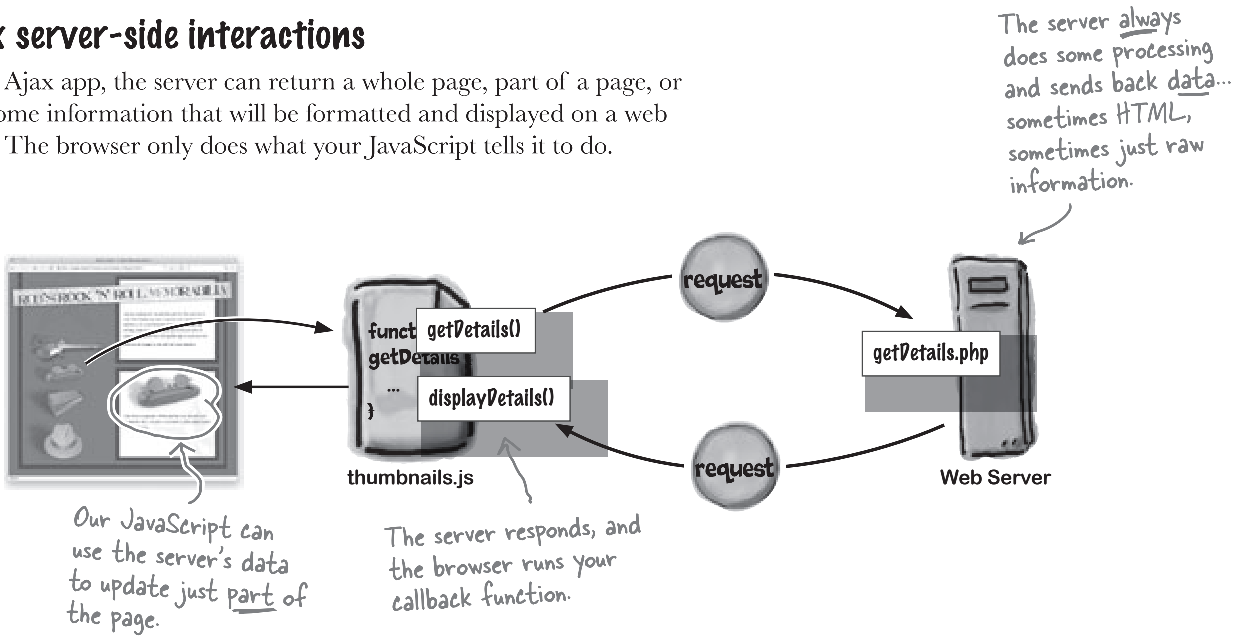
In a traditional web app, the server always responds to a request from the browser by sending back a new page. The browser throws away anything that's already displayed (including any fields the user has filled in) when that new page arrives.

Traditional server-side interactions



Ajax server-side interactions

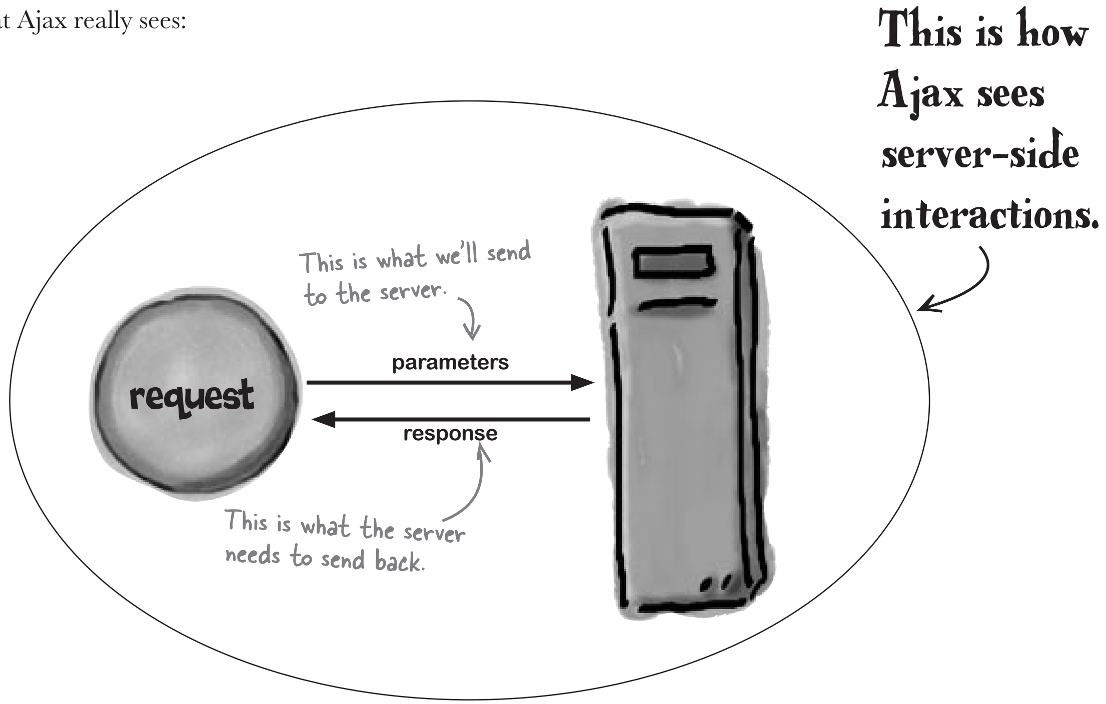
In an Ajax app, the server can return a whole page, part of a page, or just some information that will be formatted and displayed on a web page. The browser only does what your JavaScript tells it to do.



Ajax is server-agnostic

Ajax doesn't require any particular server technology. You can use Active Server Pages (ASP), PHP, or whatever you need and have access to. In fact, there's no need to get into the details of the server-side technology because ***it doesn't change how you build your Ajax apps***.

Here's all that Ajax really sees:



Sharpen your pencil

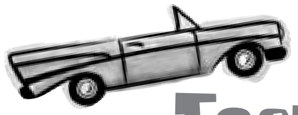
What parameter and response do we need for the interaction with the server for Rob's memorabilia page?

....

....

....

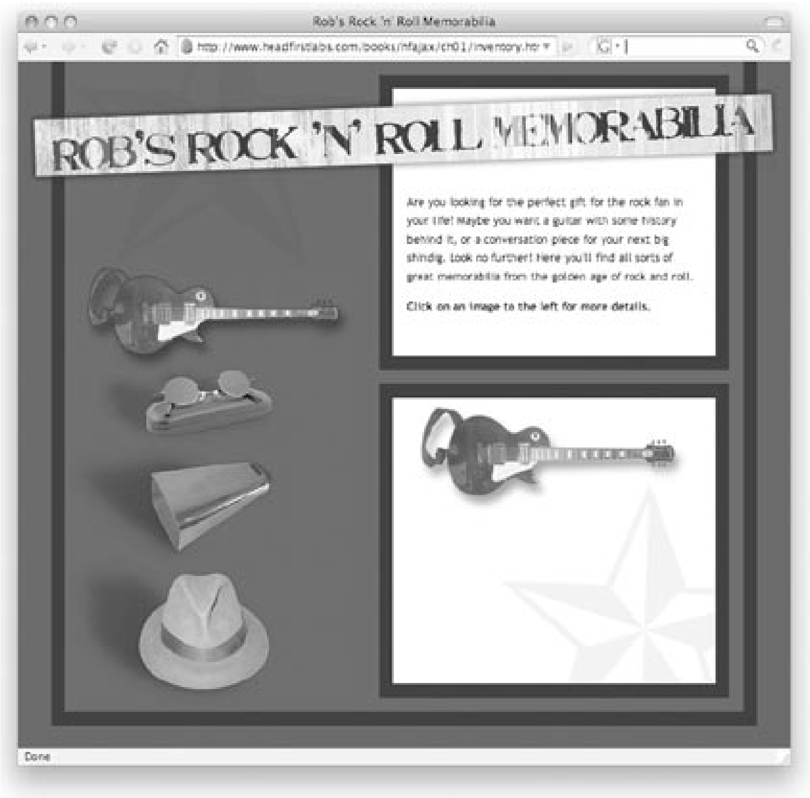
Answers on page 40.



Test Drive

Code getDetails(), and fire up your web browser.

Make sure you've got `getDetails()` coded in your `thumbnails.js` file. Load up Rob's memorabilia page, and try clicking on one of the inventory images.



BRAIN POWER

What happens? What's wrong with the page?
What do you need to do to fix the problem?

What's My Purpose?

Below on the left are several properties of the request object. Can you match each property to what it does, or what information it contains?

readyState	The status code message returned by the server, for example, "OK" for status 202.
status	Contains information sent back by the server in XML format.
responseXML	A status code returned by the server indicating, for example, success or that a requested resource is missing.
statusText	Contains textual information sent back by the server.
responseText	A number that represents the current state of the request object.

there are no Dumb Questions

Q: Can you explain what a callback function is again?

A: A callback function is a function that is executed when something else finishes. In Ajax, it's the function that's called when the server responds to a request object. The browser "calls back" that function at a certain time.

Q: So a callback executes when the server's finished with a request?

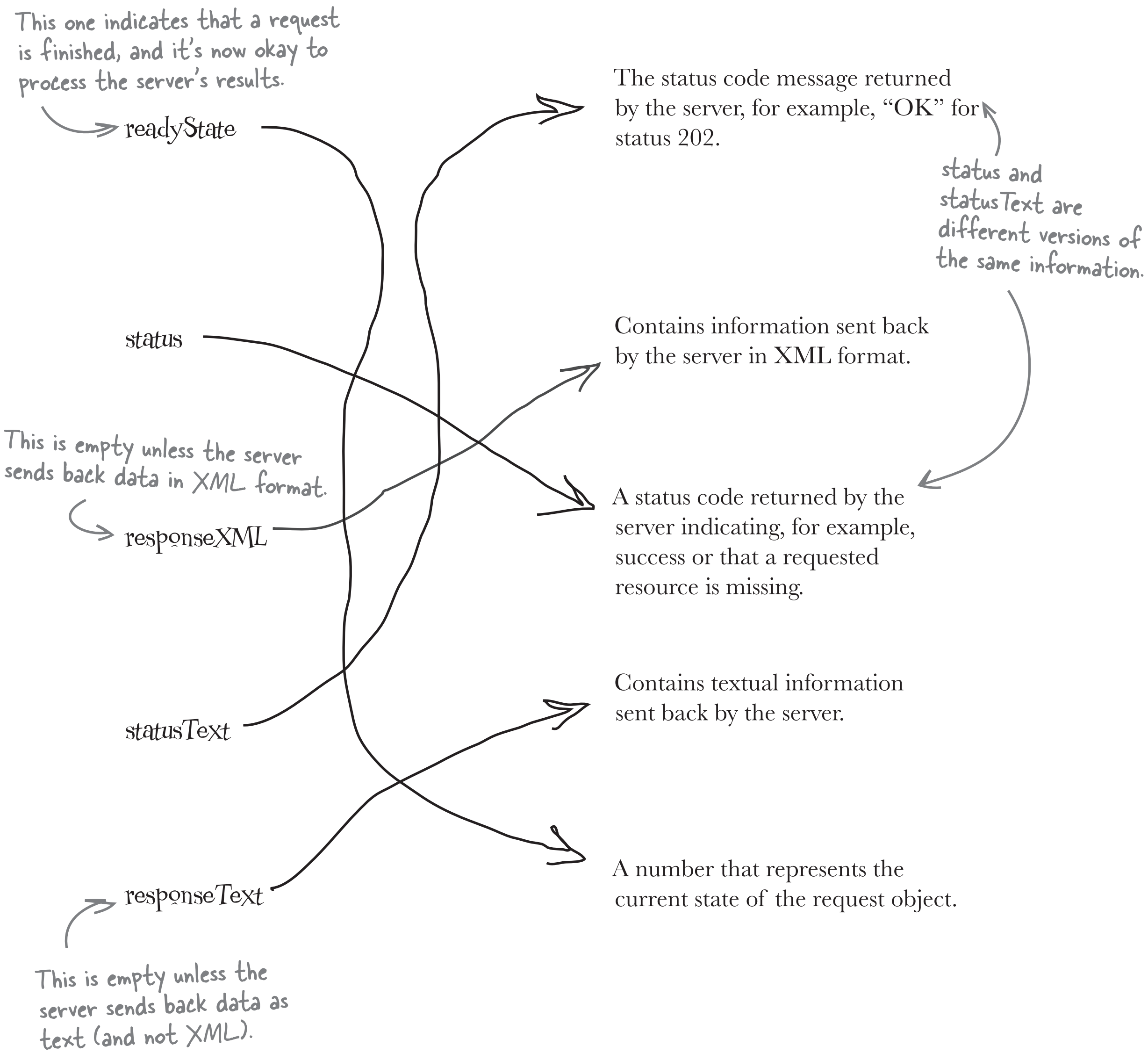
A: No, it's actually called by the browser *every time* the server responds to the request, even if the server's not totally done with the request. Most servers respond more than once to say that they've received the request, that they're working on the request, and then, again, when they've finished processing the request.

Q: Is that why the request property is called onreadystatechange?

A: That's exactly right. Every time the server responds to a request, it sets the `readyState` property of the request object to a different value. So we'll need to pay close attention to that property to figure out exactly when the server's done with the request we send it.

What's My Purpose? Solution

Below on the left are several properties of the request object. Your job was to match each property to what it does, or what information it contains.

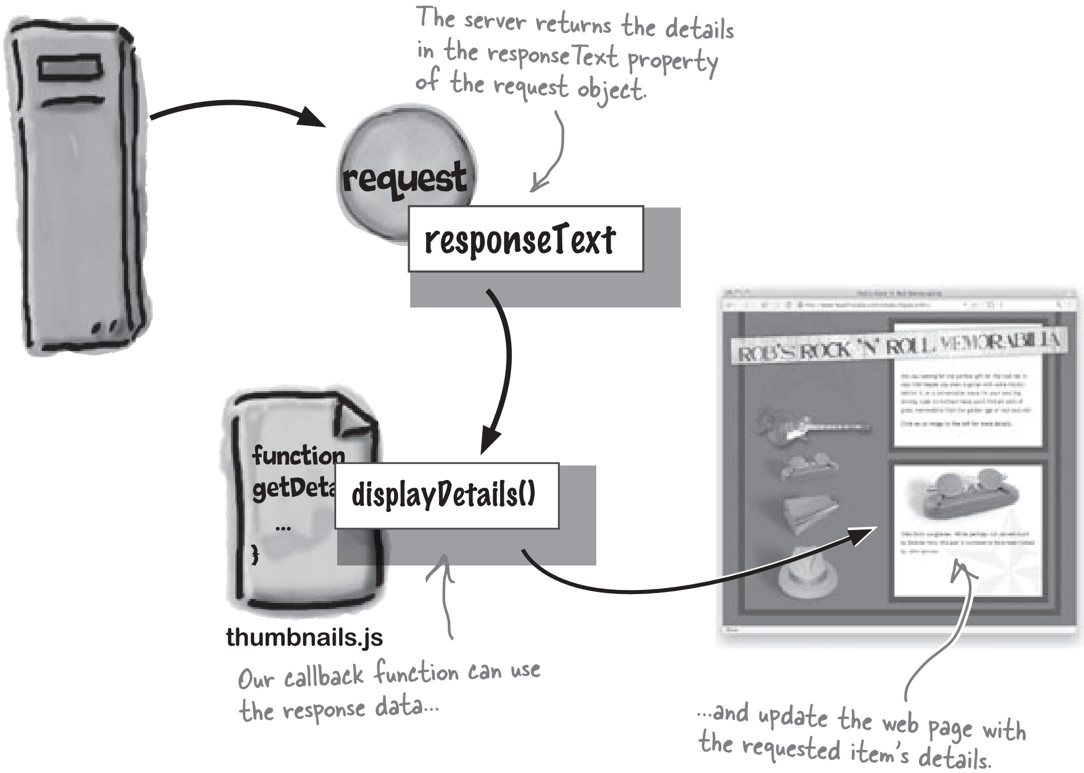


Use a callback function to work with data the server returns



How do we show the textual description for each item? Let's assume the server will send the details about an item as pre-formatted text in the `responseText` property of the request object. So we just need to get that data and display it.

Our callback function, `displayDetails()`, needs to find the XHTML element that will contain the detail information, and then set its `innerHTML` property to the value returned by the server.



there are no Dumb Questions

Q: So the server calls displayDetails() when it's finished with the request?

A: No, the *browser* actually does that. All the server does is update the `readystate` property of the request object. Every time that property changes, the browser calls the function named in the `onreadystatechange` property. Don't worry, though, we'll talk about this in a lot more detail in the next chapter.

Get the server's response from the request object's responseText property

The data we want is in the request object. Now we just need to get that data and use it. Here's what we need:

It's okay if all of this isn't completely clear to you. We'll look at ready states and status codes in a lot more detail in the next chapter.

```
function displayDetails() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      detailDiv = document.getElementById("description");
      detailDiv.innerHTML = request.responseText;
    }
  }
}
```

This line gets a reference to the XHTML element we'll put the item details in.

This line puts the XHTML returned by the server into that element.



thumbnails.js

there are no Dumb Questions

Q: What's that readyState property?

A: That's a number that indicates where the server is in its processing. 0 is the initial value, and when the server's completed a request, it's 4.

Q: So that first statement just checks to see if the server's finished with the request?

A: You got it.

Q: Why do we have to check that every time?

A: Because the browser will run your callback every time the ready state changes. Since a server might set this value to 1 when it receives the request, and to 2 or 3 as it's processing your request, you can't be sure the server's done unless `readyState` is equal to 4.

Q: And the status property?

A: That's the HTTP status code, like 404 for forbidden, and 200 for okay. You want to make sure it's 200 before doing anything with your request object.

Q: Why would the server set the ready state to 4 when the status code is something like 404?

A: Good question. We'll talk about that in the next chapter, but can you think of how a request could be complete and still have a status code that indicates a problem?

Q: Isn't innerHTML a bad thing to use?

A: It is, but sometimes it's also very effective. We'll look at better ways to change a page when we get more into the DOM in later chapters. For now, though, it works, and that's the most important thing.

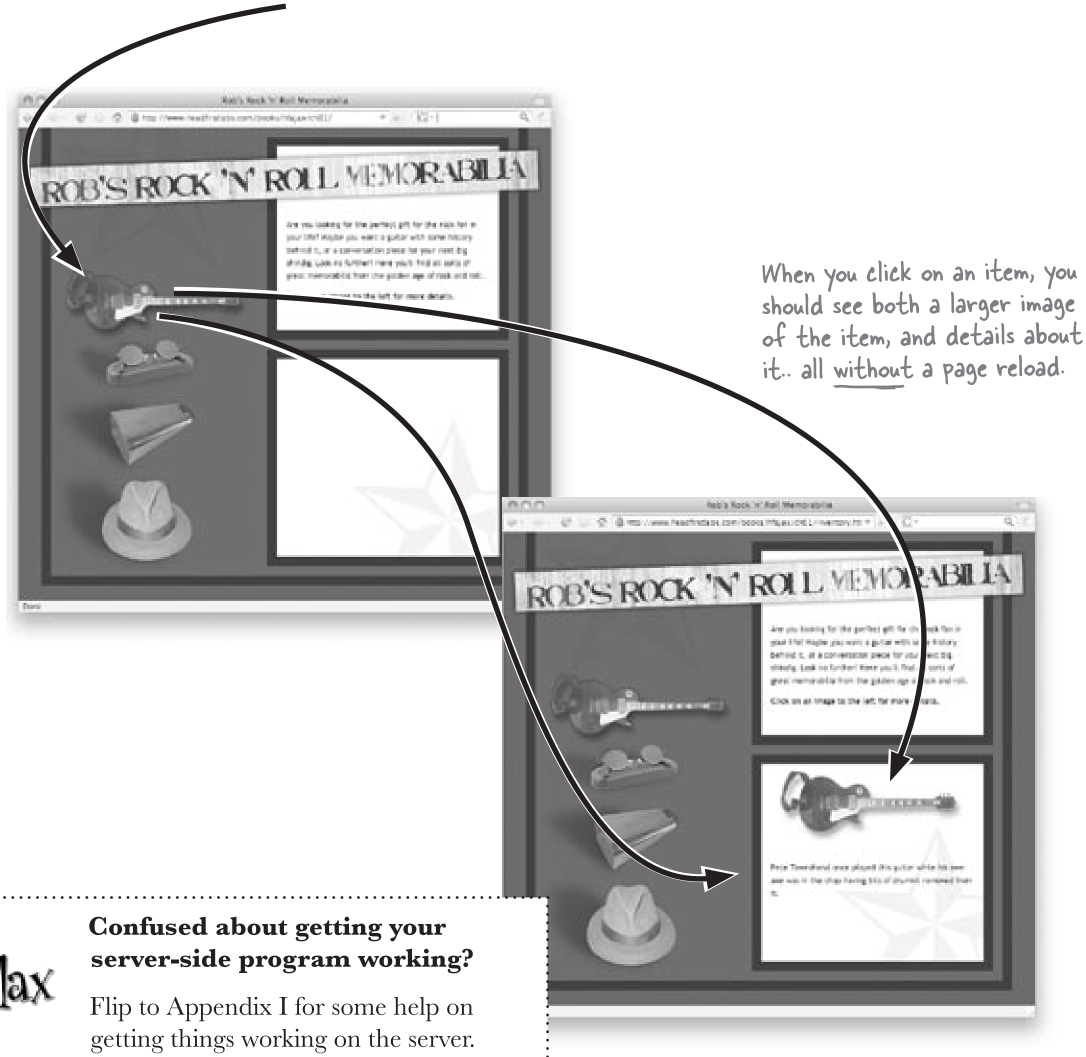
Q: Am I supposed to be getting all this? There's sure a lot going on in that callback function...

A: For now, just make sure you know that the callback is where you can use the server's response. We'll talk about callbacks, ready states, and status codes a lot more in Chapter 2.

Test Drive

Code your callback, and test out the inventory page.

Add `displayDetails()` to your `thumbnails.js` file. You should also make sure that the server-side program with the inputs and outputs detailed on page 30 is running, and that the URL in your `getDetails()` method is pointing to that program. Then fire up the inventory page and click on an item.



Confused about getting your server-side program working?

Relax

Flip to Appendix I for some help on getting things working on the server. There are also some helpful server-side resources for the book online at `http://www.headfirstlabs.com`.

Goodbye traditional web apps...

Rob's page is working more smoothly now, customers are coming back in droves, and you've helped pair vintage leather with the next-generation web.

Rob's old, traditional web app:

- ...reloaded the entire page when a user clicked on an item's thumbnail image.
- ...took a long time to load because the entire page had to be rendered by the browser on every click.
- ...felt unresponsive because the user had to wait on all those page refreshes.
- ...lost Rob business, annoyed his customers, and drained his bank account.

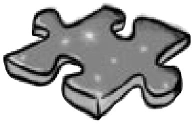
These aren't problems that just Rob's having. Almost all traditional web apps have these problems in some form or fashion.

Compare these benefits with the list on page 10... they should look pretty similar.

Rob's new, Ajax app:

- ...only changed the part of the page that needed to be updated.
- ...lets users keep viewing the page while images and descriptions are loaded behind the scenes, asynchronously.
- ...reduced the need for his users to have super-fast connections to use his site.





AjaxAcrostic

Take some time to sit back and give your right brain something to do. Answer the questions in the top, then use the letters to fill in the secret message.

This is the language you use to script Ajax pages.

___	___	___	___	___	___	___	___	___	___
1	2	3	4	5	6	7	8	9	10

This type of function gets called when a process completes.

___	___	___	___	___	___	___	___
11	12	13	14	15	16	17	18

This request object property tells us when the server has finished processing.

___	___	___	___	___	___	___	___	___	___
19	20	21	22	23	24	25	26	27	28

If something goes wrong at the server, this property will tell us what.

___	___	___	___	___	___
29	30	31	32	33	34

The browser will put text that the server returns in this property.

___	___	___	___	___	___	___	___	___	___	___	___
35	36	37	38	39	40	41	42	43	44	45	46

If there's a problem, we can get a description of it in this property.

___	___	___	___	___	___	___	___	___	___
47	48	49	50	51	52	53	54	55	56

Use the letters from the blanks above to fill in these...

___	___	___	___		___	___	___	___		___	___	___
49	1	31	45		13	54	10	29		23	39	33

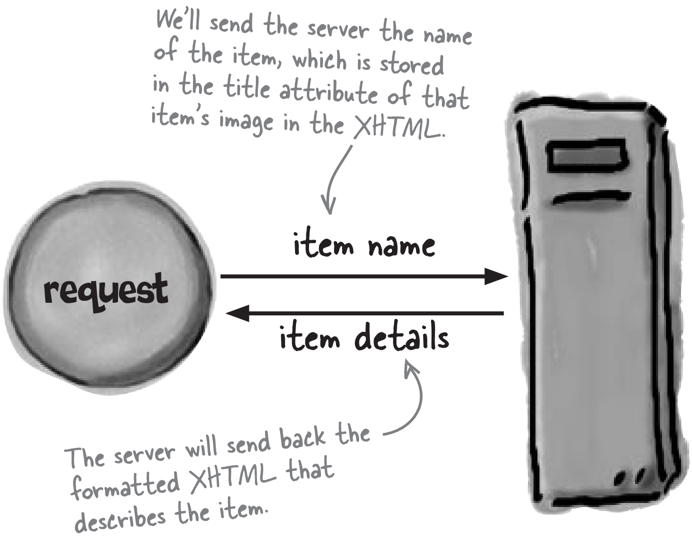
___	___	___	___	___		___	___	___	___	___	___	___	___	___	___
15	51	8	14	22		19	28	37	9	39	40	34	8	3	44

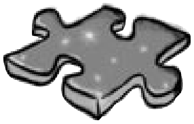
___	___	___	___	___	___	___	___	___	___	___	___
31	9	38	14	8	6	26	46	8	39	40	24

Sharpen your pencil Solution

From page 31

What parameter and response do we need to implement Rob's page?





AjaxAcrostic Solution

Take some time to sit back and give your right brain something to do. Answer the questions in the top, then use the letters to fill in the secret message.

This is the language you use to script Ajax pages.

J	A	V	A	S	C	R	I	P	T
1	2	3	4	5	6	7	8	9	10

This type of function get called when a process completes.

C	A	L	L	B	A	C	K
11	12	13	14	15	16	17	18

This request object property tells us when the server has finished processing.

R	E	A	D	Y	S	T	A	T	E
19	20	21	22	23	24	25	26	27	28

If something goes wrong at the server, this property will tell us what.

S	T	A	T	U	S
29	30	31	32	33	34

The browser will put text that the server returns in this property.

R	E	S	P	O	N	S	E	T	E	X	T
35	36	37	38	39	40	41	42	43	44	45	46

If there's a problem, we can get a description of it in this property.

S	T	A	T	U	S	T	E	X	T
47	48	49	50	51	52	53	54	55	56

A	J	A	X		L	E	T	S		Y	O	U
49	1	31	45		13	54	10	29		23	39	33

B	U	I	L	D		R	E	S	P	O	N	S	I	V	E
15	51	8	14	22		19	28	37	9	39	40	34	8	3	44

A	P	P	L	I	C	A	T	I	O	N	S
31	9	38	14	8	6	26	46	8	39	40	24

2 designing ajax applications

Thinking Ajaxian



Welcome to Ajax apps—it's a whole new web world.

So you've built your first Ajax app, and you're already thinking about how to change all your web apps to make requests asynchronously. But that's not all there is to Ajax programming. You've got to ***think about your applications differently***. Just because you're making asynchronous requests, doesn't mean your application is user-friendly. It's up to you to help your users **avoid making mistakes**, and that means **rethinking** your entire application's **design**.

Mike's traditional web site ~~sucks~~

Note from HR: Can we use a less offensive term? How about "consistently annoys every one of Mike's users"?

Mike's got the hippest movie reviews going, and he's taking his popular opinions online. Unfortunately, he's having problems with his registration page. Users visit his site, select a username, type in a few other details, and submit their information to get access to the review site.

The problem is that if the username's taken, the server responds with the initial page again, an error message... and none of the information the user already entered. Worse, users are annoyed that after waiting for a new page, they get nothing back but an error message. They want movie reviews!

Users shouldn't have to fill out eight fields to find out if the data in the <u>first</u> field is valid.



Right now the user fills out the form and clicks the "Register" button... and then waits, and hopes for the best.

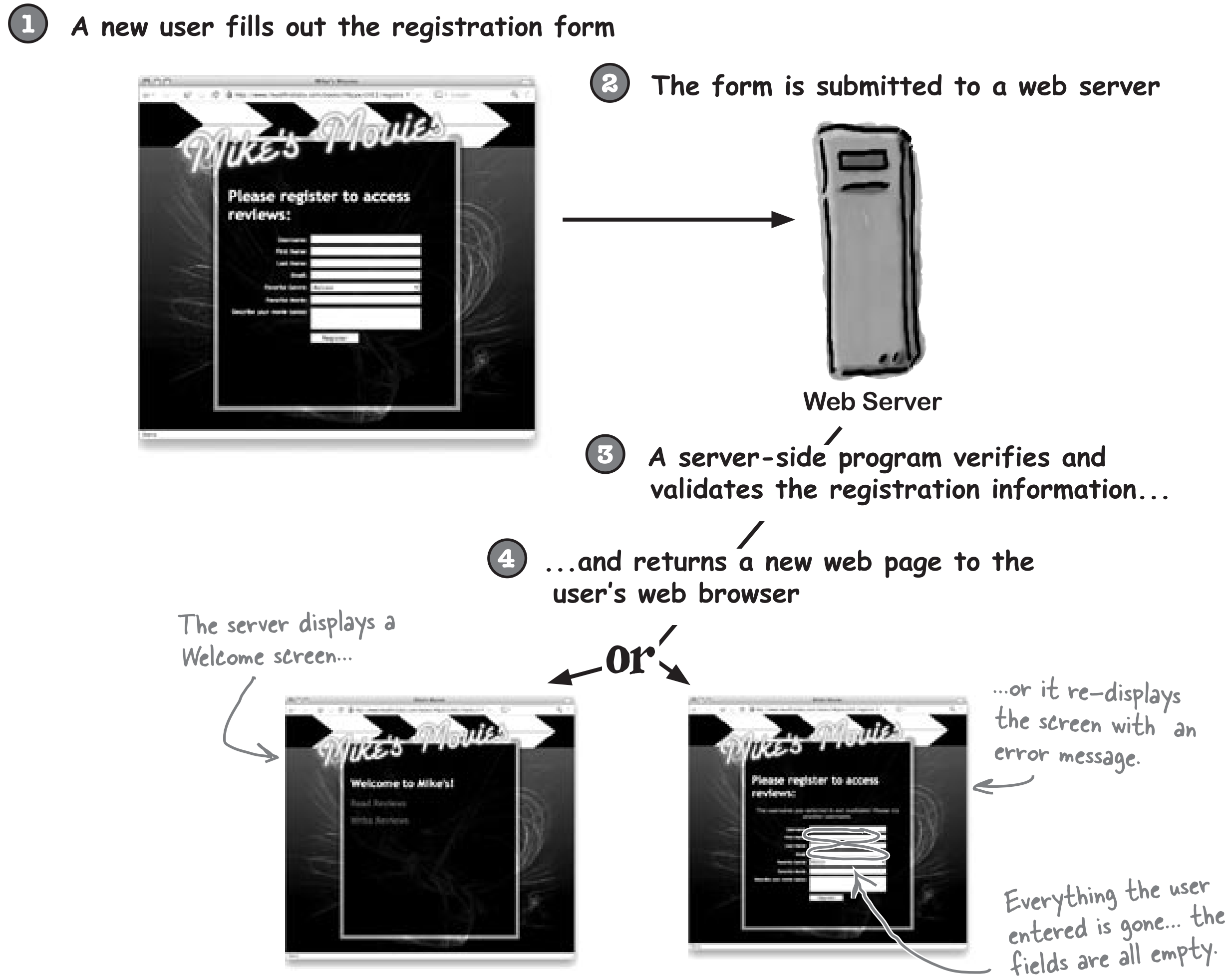
I'm getting tons of registrations, so lots of usernames are already taken. This is how everyone else handles signups, but I'm getting flooded with complaints. Can you help me out?

Mike's movie review site looks great and has tons of terrific reviews... but only if users can get signed up and past the registration page.



Exercise

Mike's got real problems, but with one Ajax app under your belt, you should probably have some ideas about what Mike needs. Take a look at the diagram of what happens with Mike's app now, and make notes about what you think *should* happen. Then, answer the questions at the bottom of the page about what you'd do to help Mike out.



What do *you* think is the single biggest problem with Mike's site? ....

....

....

What would *you* do to improve Mike's site? ....

....

....

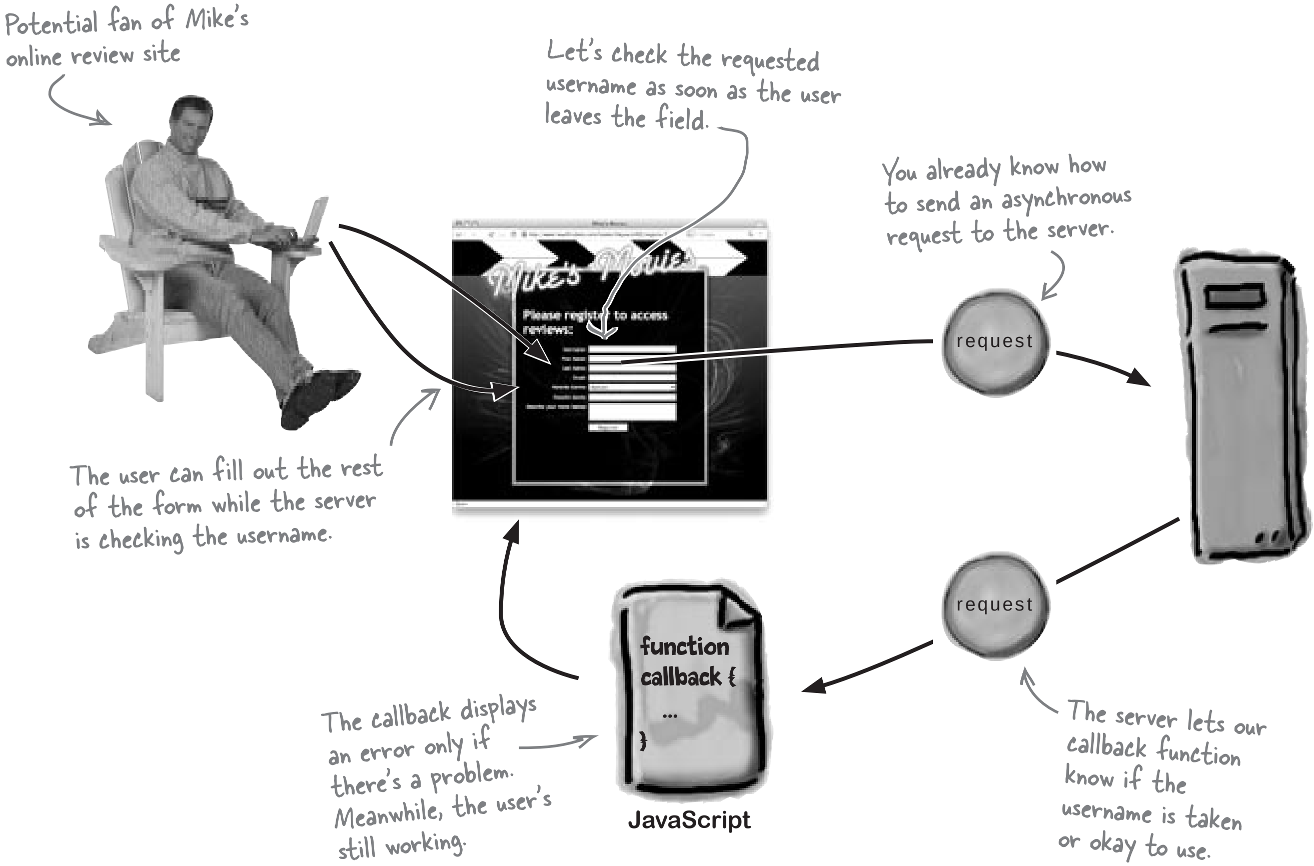
Let's use Ajax to send registration requests ASYNCHRONOUSLY

Ajax is exactly the tool you need to solve the problem with Mike's page. Right now the biggest problem is that users have to wait for a full page refresh to find out their requested username is already taken. Even worse, if they need to select a different username, they've got to re-type all their other information again. We can fix both of those problems using Ajax.

Did you write down something similar to this as Mike's biggest problem?

We'll still need to talk to the server to find out whether a username has been taken, but why wait until users finish filling out the entire form? As soon as they enter a username, we can send an **asynchronous request** to the server, check the username, and report any problems directly on the page—all ***without*** any page reloads, and ***without*** losing the user's other details.

It's okay if you didn't think about sending the request as soon as the user types in their username... but bonus credit if you did!



All this just so some movie buff doesn't have to retype their name and email address? Doesn't this seem like a bit of overkill?



Don't annoy your users... ever!

On the Internet, your competitors are only a click away. If you don't tell your users about a problem right away, or if you ever make them re-do something, you're probably going to lose them forever.

Mike's site may not be a big moneymaker (yet), or even seem that important to you... but it might to his fans. One day a user you're helping him not annoy may land him a six-figure income writing movie reviews for the New York Times. But Mike won't ever know if his site is hacking his users off. That's where your Ajax skills can help.

Important Ajax design principle

Don't annoy your users



If there's a problem with your web app, let your users know about it as quickly and clearly as possible. And you should never throw away anything the user has already done, even if something happened that they (or you) weren't expecting.

there are no Dumb Questions

Q: That design principle isn't really Ajax-specific, is it?

A: Nope, it applies to all web applications, ... in fact, to all types of applications. But with Ajax apps, especially asynchronous requests, lots of things can go wrong. Part of your job as a good Ajax programmer is to protect your users from all those things, or at least let them know what's going on if and when they do happen.

Exercise

It's time to get to work on Mike's site. Below are 5 steps that you'll need to execute to get his site working, but the details about each step are missing, and the ordering is a mess. Put the steps in order, and write a sentence or two about exactly what should happen on each step.

? Create and configure a new request object

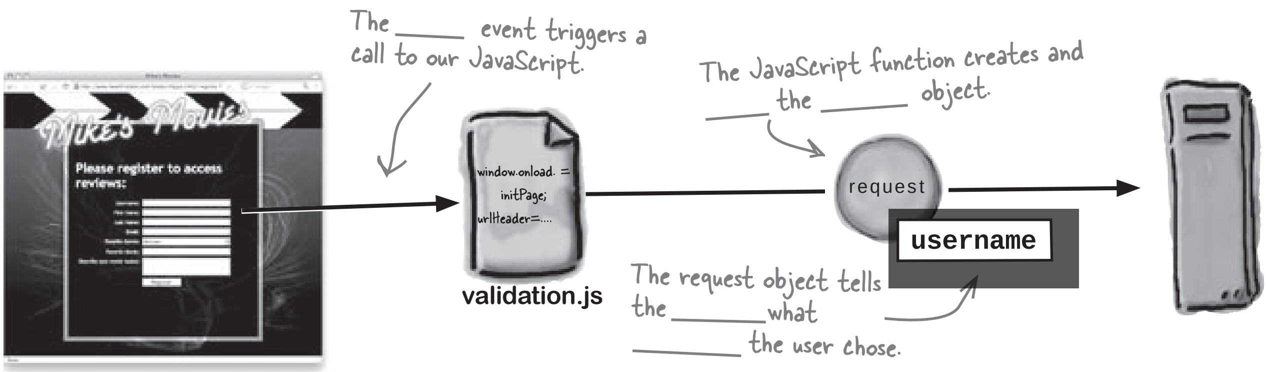
? Set up event handlers for the web form's fields

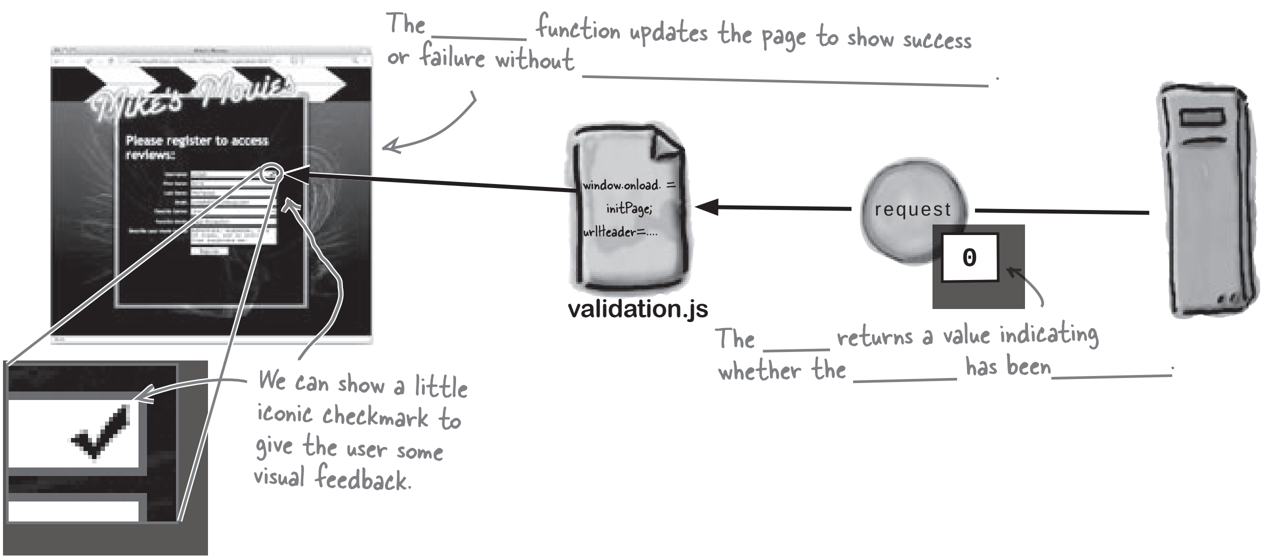
? Verify the requested username

? Report any problems with the requested username

? Update the registration page's XHTML and CSS

After you've got your steps in order, take a look at the two diagrams below that describe some of the interactions in an Ajax version of Mike's app. See if you can fill in the blanks so that the diagrams are complete and the annotations are accurate.







Your job was to order the steps to build an Ajax-version of Mike's movie review site, and fill in the missing descriptions of each step. You also should have filled in the missing words in the diagrams.

1 Update the registration page's XHTML and CSS

We'll need to add `<script>` elements to the registration form to reference the JavaScript code we'll be writing.

Technically you can write the code for these steps in any order, but this is the flow that the app will follow and that we'll use to update Mike's app in this chapter.

2 Set up event handlers for the web form's fields

We'll need some initiational code to set up an `onblur` event for the username field on the page. So when the user leaves that field, we'll start the request process.

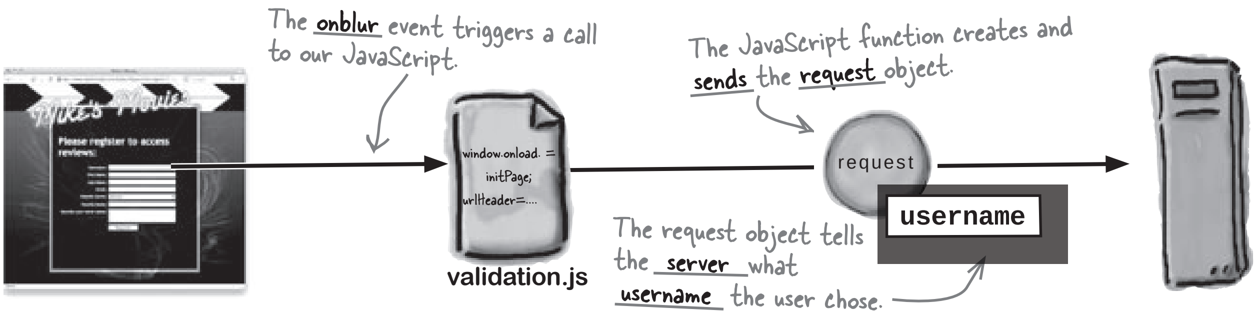
3 Create and configure a new request object

We can use the same `createRequest()` function from Chapter 1 to create the request, and then we'll add the user's requested username to the URL string to get that over to the server.

We skimmed this function in the last chapter, but we'll look at it in detail in this chapter.

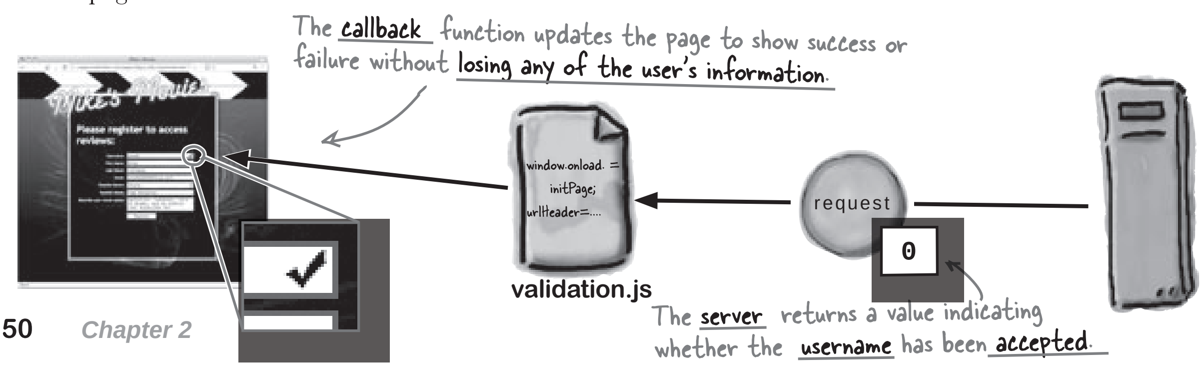
4 Verify the requested username

Once we've created a request object, we need to send it off to the server to make sure that the requested username hasn't been taken by someone else. We can do this asynchronously, so the user can keep filling in the page while the server's checking on their username.



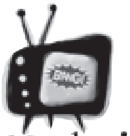
5 Report any problems with the requested username

When the request object returns, the callback function can update the page to show whether the username check succeeded or failed.



Update the registration page

The basic structure of Mike's registration page is already in place, so let's go ahead and add a `<script>` tag to load the JavaScript we'll write. Then, we can set up the username field on the web form to call a JavaScript function to make a request to the server.



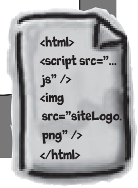
Watch it!

Use an opening and closing <script> tag.

*Some browsers will error out if you use a self-closing <script> tag, like <script />. Always use **separate** opening and closing tags for <script>.*

```
<head>
    <title>Mike's Movies</title>
    <link href="movies.css" rel="stylesheet" type="text/css" />
    <script src="scripts/validation.js" type="text/javascript"></script>
</head>
```

Just like in the last chapter, we'll write validation.js as we go through the chapter.



registration.html

Make these changes in registration.html, Mike's registration page.



Download the registration page's XHTML and CSS.

If you haven't already done so, download the sample files for the chapter from **`www.headfirstlabs.com`**. Look in the Chapter2 folder for the file named **`registration.html`**, and then add the script tag shown in bold.

there are no Dumb Questions

Q: What's the big deal? This is all just like the rock and roll site from last chapter, isn't it?

A: So far, it is. But most Ajax apps start with a few `<script>` tags and some external JavaScript files.

Q: But we're still just sending a request and getting a response, right?

A: Sure. In fact, almost all Ajax apps can be described that simply. But as you'll see as we get into the registration page, there are actually two interactions possible: the one we're building to check a username, and the Submit button the user will press when they've filled out the form.

Q: What's the big deal about that?

A: What do you think? Can you see any problems with having two ways of making two different requests to a web server?

Hey, there's more to do in that XHTML. What about the onblur event handler on the username field? We want to run some code every time the user enters a username, right?



Separate your page's content from its behavior.

We could call the JavaScript directly from the XHTML by, for example, putting an `onblur` event in the username form field. But that's mixing the content of our page with its behavior.

The XHTML describes the ***content*** and ***structure*** of the page: what data is on the page, like the user's name and a description of the movie review site, and how it's organized. But how a page reacts to the user doing something is that page's ***behavior***. That's usually where your JavaScript comes in. And the CSS defines the presentation of your page: how it looks.

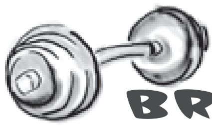
Keeping content, behavior, and presentation separate is a good idea, even when you're building a relatively simple page all by yourself. And when you're working on complex applications that involve a lot of people, it's one of the best ways to avoid accidentally messing up somebody else's work.



Separate your page's content, behavior, and presentation.

Whenever possible, try to keep your page's content (the XHTML) separate from its behavior (JavaScript and event handlers) and its presentation (the CSS look-and-feel). Your sites will be more flexible and easier to maintain and update.

You'll hear some people refer to this principle as unobtrusive JavaScript.



BRAIN BARBELL

How do you think separating the content of a site from its presentation and behavior makes it easier to change?



Event Handlers Exposed

This week's interview:
Where are you really from?

Head First: It's good to have you with us, Event Handler. We've got some really interesting questions for you this week.

Event Handler: Really? I'm always eager to respond to questions.

Head First: Actually, there's this one question that everyone's been asking. Where exactly are you from?

Event Handler: Well, I hail from the land of ECMA, which was—

Head First: Oh, no, I mean, where are you ***called*** from?

Event Handler: Hmm... Well, I think the ECMA folks might want their story told, but if you insist... I usually get called from an XHTML form field or a button, things like that. Sometimes from windows, too.

Head First: So you're called from XHTML pages?

Event Handler: Most of the time, that's right.

Head First: That's what I thought. Well, that settles the dispute. You all heard it here first—

Event Handler: Wait, wait! What dispute?

Head First: Well, we had JavaScript calling in, swearing he could call you. Something about behavior calling behavior... it was really just nonsense.

Event Handler: Oh, you must be talking about assigning me programmatically. Very smart, that JavaScript...

Head First: Programmatically? What does that mean?

Event Handler: You see, I'm really just a property at heart—

Head First: Uh oh, is this more about ECMA?

Event Handler: —that can be set with JavaScript. No, now listen. You know about the DOM, right?

Head First: Well, not really... isn't that a later chapter?

Event Handler: Never mind. Look, everything on a web page is just an object. Like fields and buttons, they're just objects with properties.

Head First: Okay, sure, we've met some fields before. Nice folks. But Button, he never would return our calls...

Event Handler: Well, anyway, events like `onblur` or `onload` are tied to me through those properties.

Head First: You mean, like in XHTML when you say `onblur="checkUsername()"` on an input element?

Event Handler: Exactly! It's just a property of the input field. You're just telling the browser what function to run... you know, how to handle that event.

Head First: I'm totally lost...

Event Handler: Well, you can use JavaScript to assign a value to a property of an object, right?

Head First: So you're saying that you don't have to just assign event handlers from an XHTML page?

Event Handler: Right! You can do it directly in JavaScript code... and keep your content and structure separate from your behavior.

Head First: Well, this is quite surprising. But how do you get your JavaScript to run in the first place to assign an event handler?

Event Handler: Well, that's the trick. Any ideas?

Head First: I'm not sure. Let's ask our audience...

How can you get an initial piece of JavaScript to run *without* referencing a function in your XHTML page?

....

....

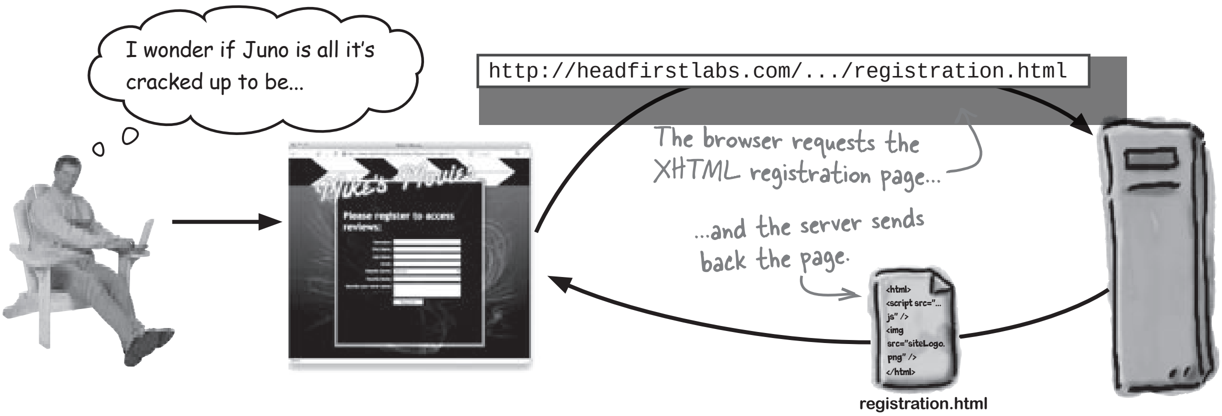
....

Set the window.onload event handler... PROGRAMMATICALLY

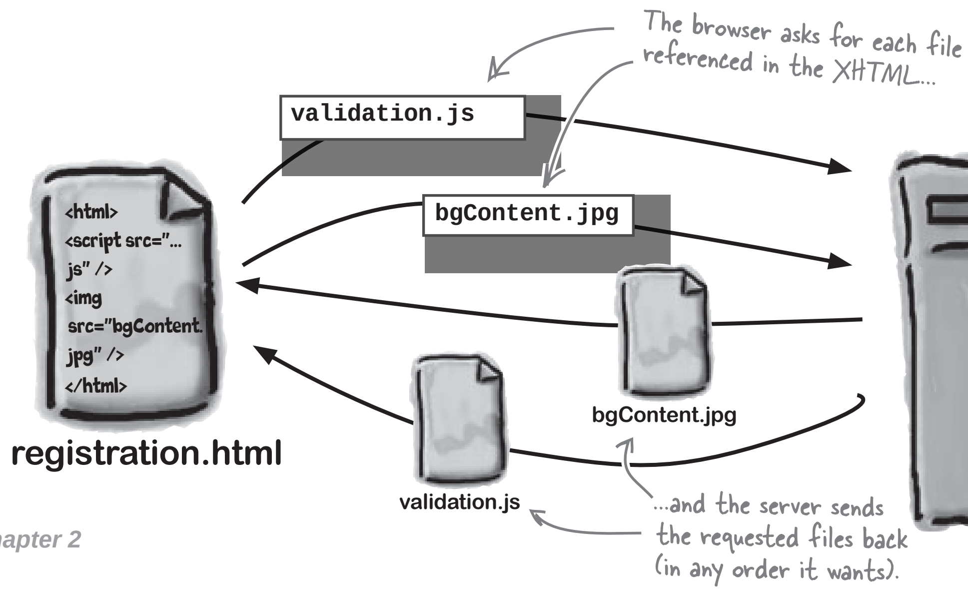
We want some JavaScript code to run when the registration page loads, and that means attaching that code as the event handler on one of the first page events, `window.onload`.

And we can do that programmatically by setting the `onload` property of the `window` object. But how do we do that? Let's look at exactly what happens when the registration page is requested by a user visiting Mike's movie review site:

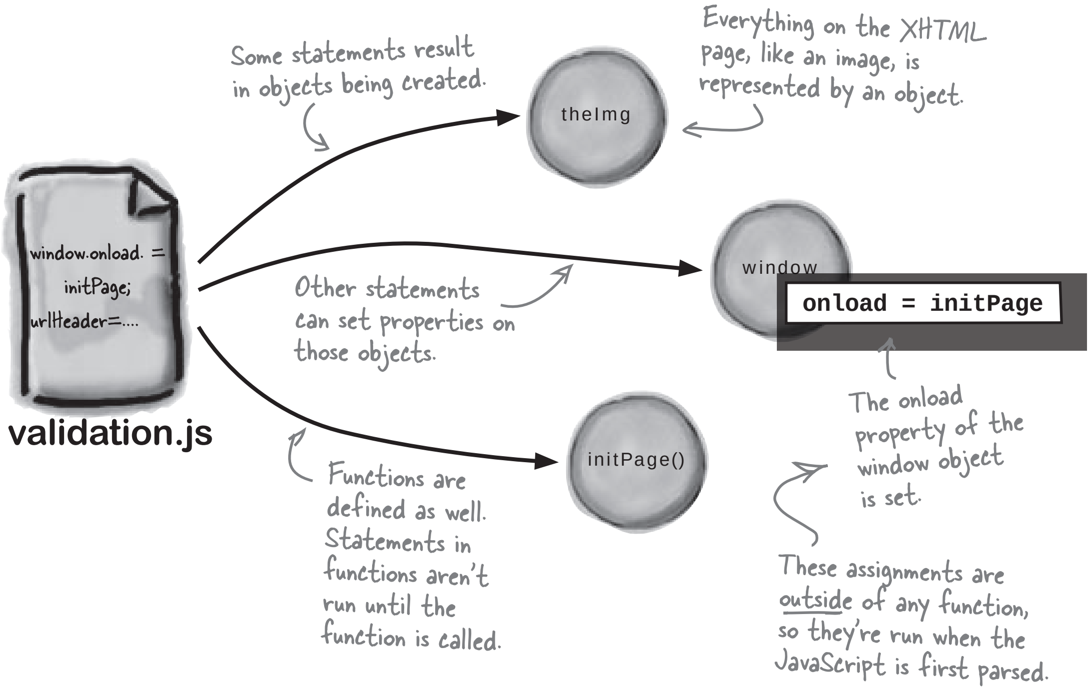
First, a user points their browser at Mike's registration page.



Then, the browser starts parsing the page, asking for other files as they're referenced.

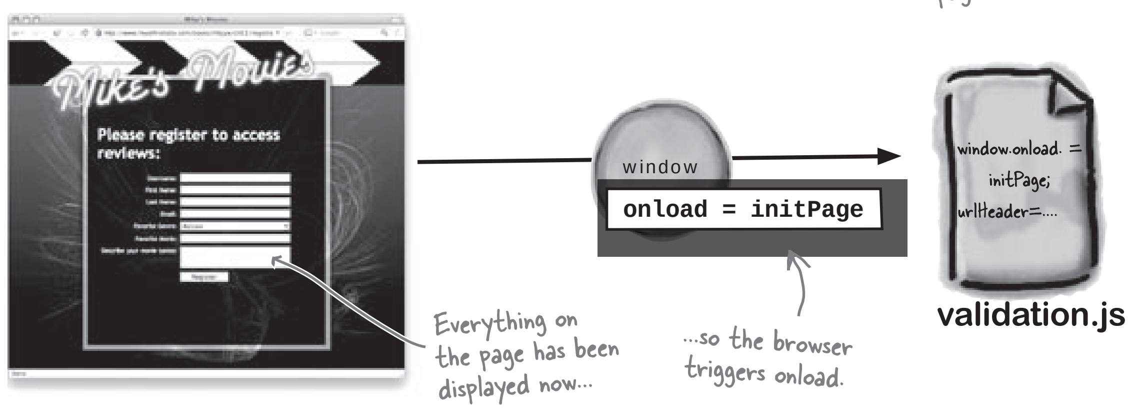


If the file is a script, the browser parses the script, creates objects, and executes any statements not in a function.



Finally, after all referenced files are loaded and parsed, the browser triggers the window.onload event and calls any function that's registered to handle that event.

All of this happens before you can actually use the page... so it's lightning fast!



Code in your JavaScript outside of functions runs when the script is read



validation.js

We want to set an event handler up to run as soon as a user loads the registration page. So we need to assign a function to the `onload` property of the `window` object.

And to make sure this event handler is assigned as soon as the page loads, we just put the assignment code outside of any functions in `validation.js`. That way, before users can do anything on the page, the assignment happens.

This code isn't in a function... it runs as soon as the script file is read by the web browser.

This line tells the browser to call the initPage function as soon as the elements on the page have been loaded.

```
window.onload = initPage;

function initPage(){
  document.getElementById("username").onblur =
    checkUsername;
}

function checkUsername() {
  // get a request object and send
  //   it to the server
}

function showUsernameStatus() {
  // update the page to show whether
  //   the user name is okay
}
```

This tells the browser to call the checkUsername() function when the user leaves the username field on the form.

Here's another case where we're assigning an event handler programmatically.

We'll look at getElementByID in detail in Chapters 5 and 6. For now, you only need to understand that it returns an element in the XHTML page with the specified id.

This is the function that will create and send the request object. We'll build this a little later.

This will update the page after the browser gets a response from the server.



Create the initial version of validation.js.

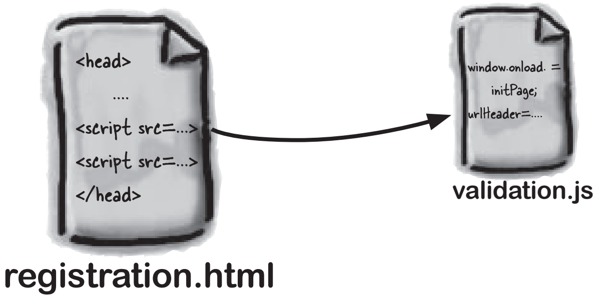
Create a new file called **`validation.js`** in a text editor, and add the function declarations shown above. Remember to assign the `initPage()` function to the `window` object's `onload` property!

What happens when...

There's a lot going on in this step. Let's go through it to make sure everything's happening exactly when we want it to.

First...

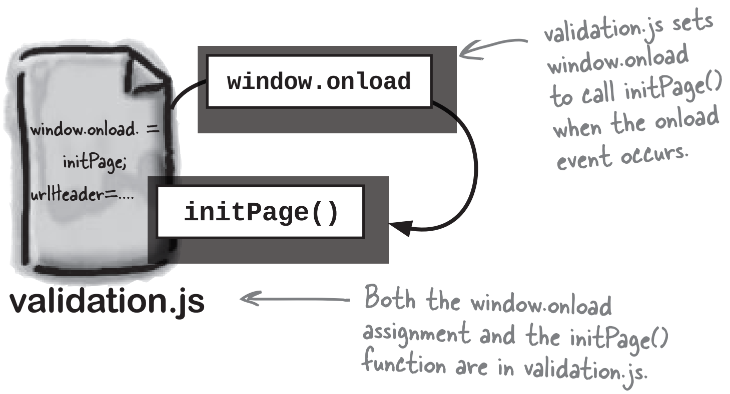
When the browser loads the XHTML file, the `<script>` tag tells it to load a JavaScript file. Any code that's outside of a function in that script file will be executed *immediately*, and the browser's JavaScript interpreter will create the functions, although the code inside those functions won't run yet.



...and then...

The `window.onload` statement isn't in a function, so it will be executed as soon as the browser loads the `validation.js` script file.

The `window.onload` statement assigns the `initPage()` function as an event handler. That function will be called as soon as all the files the XHTML refers to have been loaded but before users can use the web page.

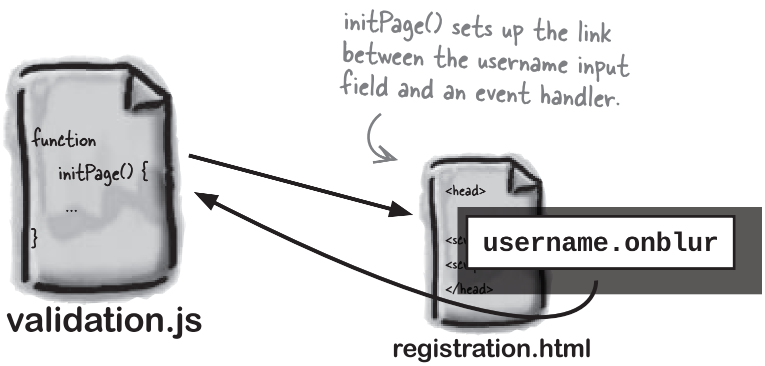


Even though these happen in sequence, ALL of this occurs before users can interact with the web page.

...and finally...

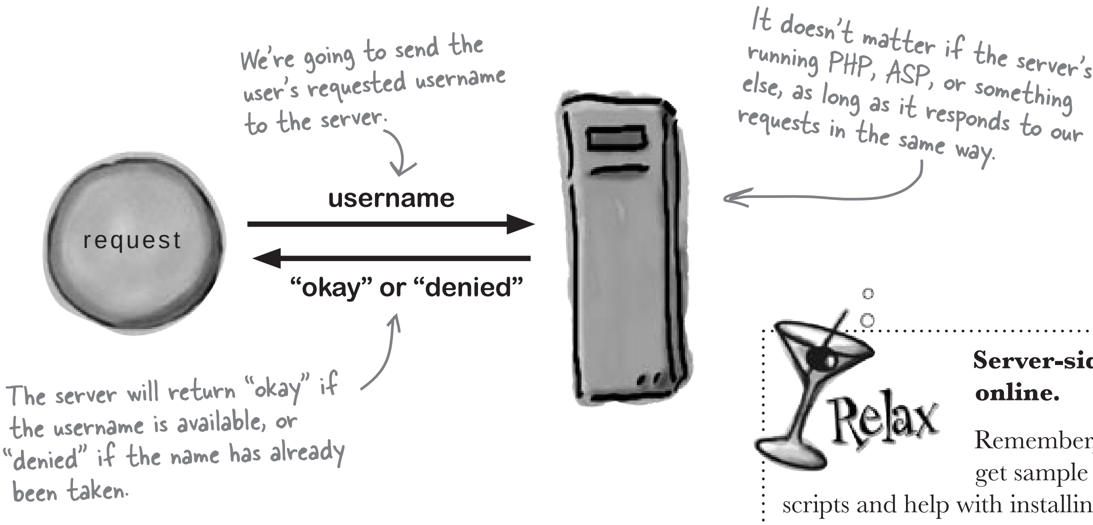
The `initPage()` function runs. It finds the field with an id of "username." Then, it assigns the `checkUsername()` function to the `onblur` event of that field.

This is the same as putting `onblur="checkUsername()"` in the XHTML. But our way is cleaner because it separates the code (the JavaScript function) from the structure and content (the XHTML).



And on the server...

Before we can test out all our work on Mike's registration page, we need to check out the server. What does the server need to get from our request? What can we expect from the server?



Relax

Server-side help is online.

Remember, you can get sample server-side scripts and help with installing them online at **`http://www.headfirstlabs.com`**.

there are no Dumb Questions

Q: What's that window object again?

A: The window object represents the user's browser window.

Q: So window.onload runs as soon as the user requests a page?

A: Not quite that fast. First, the browser parses the XHTML and any files referenced in the XHTML, like CSS or JavaScript. So code in your scripts outside of functions is run before the function specified in the `window.onload` event.

Q: And that's why I can assign a function to window.onload in my script file?

A: Exactly. Any scripts referenced in your XHTML page are read *before* the `onload` event triggers. Then, after `onload` triggers, users can actually use your page.

Q: I thought you had to call JavaScript code to get it to run. What gives?

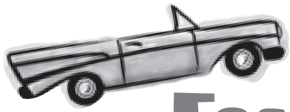
A: Good question. You have to call code in JavaScript *functions* to get it to run. But any code that's *not* in a function gets run as soon as the browser parses that line of code.

Q: But we should probably test this and make sure it works, right?

A: Right. Always test your application designs before you assume they're working.

Q: But nothing happens in this code. How do I test it?

A: That's another good question. If you have code that doesn't produce a visible result, you may want to resort to the trusty `alert()` function...



Test Drive

Take the new registration page for a spin.

Make sure you've made all the changes to `registration.html` and `validation.js`, and then load the registration page up in your browser. Doesn't look much different, does it?

The `initPage()` function doesn't do anything visible, and `checkUsername()` function doesn't do anything at all yet... but we still need to make sure `checkUsername()` is actually called when users enter a username and go to another field.

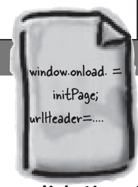
It's a bit of a hack, but let's add some `alert()` statements to our code to make sure the functions we've written are actually getting called:

```
window.onload = initPage;

function initPage(){
  document.getElementById("username").onblur = checkUsername;
  alert("Inside the initPage() function");
}

function checkUsername() {
  // get a request object and send it to the server
  alert("Inside checkUsername()");
}

function showUsernameStatus() {
  // update the page to show whether the username is okay
}
```



validation.js

Now try things out!

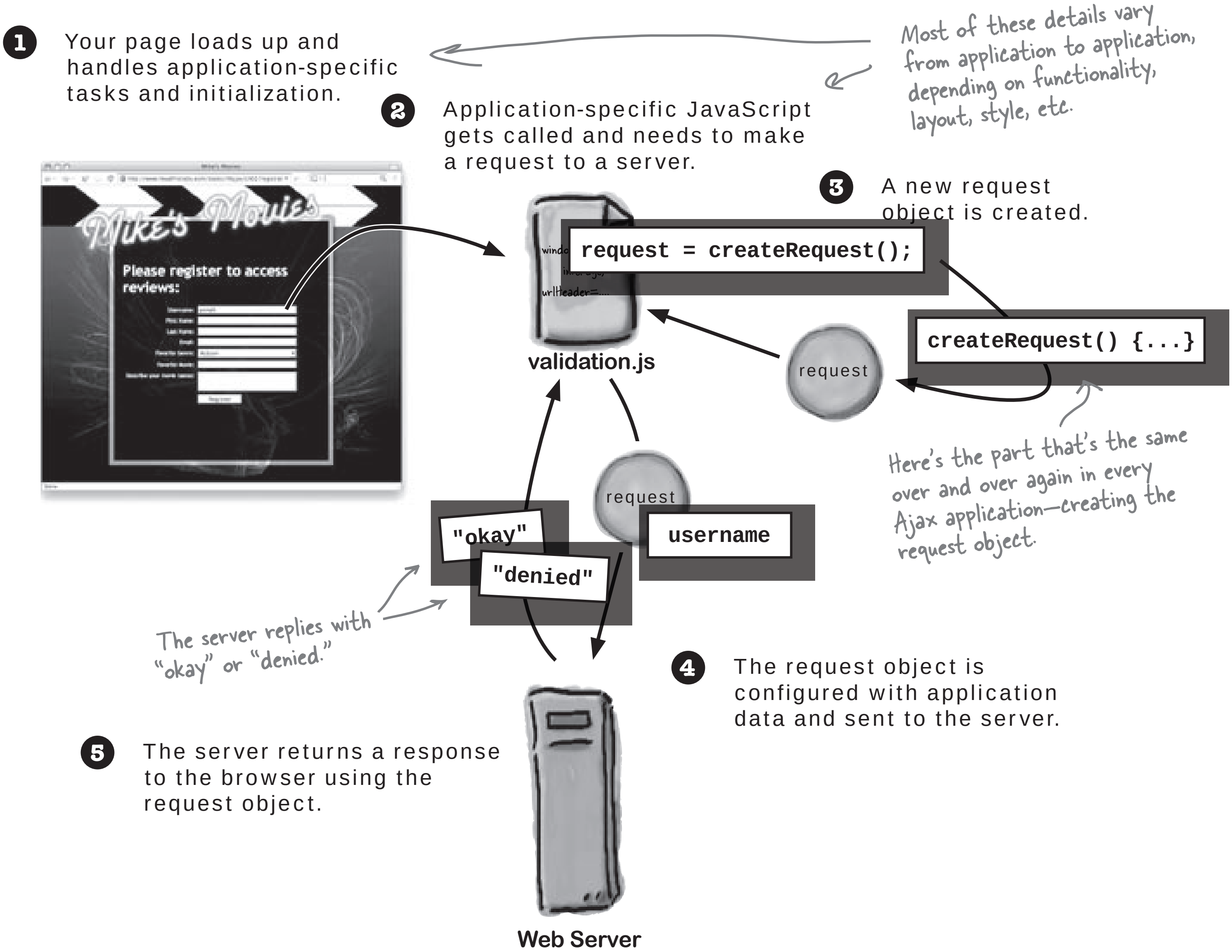


Some parts of your Ajax designs will be the same... every time

We've already used `window.onload` and an `initPage()` function twice: once for Rob's rock and roll store, and again for Mike's registration page. Next up is creating a request object that works the same for the registration page as it did for Rob's rock and roll site.

In fact, lots of things in Ajax apps are the same. Part of your job, though, is to build code so you don't have to write those same bits of code over and over again. Let's see how creating and using a request object looks in Mike's movie review site:

Good application designers look for similarities and find ways to reuse code from other designs and applications.



createRequest() is always the same

We need a function to create a request object in almost every Ajax application... and we've already got one. It's the `createRequest()` function you saw back in Chapter 1, in fact. Let's take a closer look at how this function creates a request in all types of situations, with all types of client browsers.

Watch it!

IE 5 on the Mac still doesn't work, even with this browser-independent code.

For this to be reusable, it can't depend on a certain browser or application-specific details.

```
function createRequest() {
  try {
    request = new XMLHttpRequest();
  } catch (tryMS) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (otherMS) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = null;
      }
    }
  }

  return request;
}
```

This handles lots of browsers and, therefore, lots of different users.

Remember, we have to keep trying until we find a syntax that the browser understands.

This line sends the request back to the calling code.

there are no Dumb Questions

Q: So what is this request object thing really called?

A: Most people call it an XMLHttpRequest, but that's a real mouthful. Besides, some browsers call it something different, like XMLHTTP. It's really easier to simply refer to it as a request object, and avoid being too browser-specific. That's how most everyone thinks about it anyway: as a request

Wait... If it's exactly the same code as before, why can't we just copy and paste?

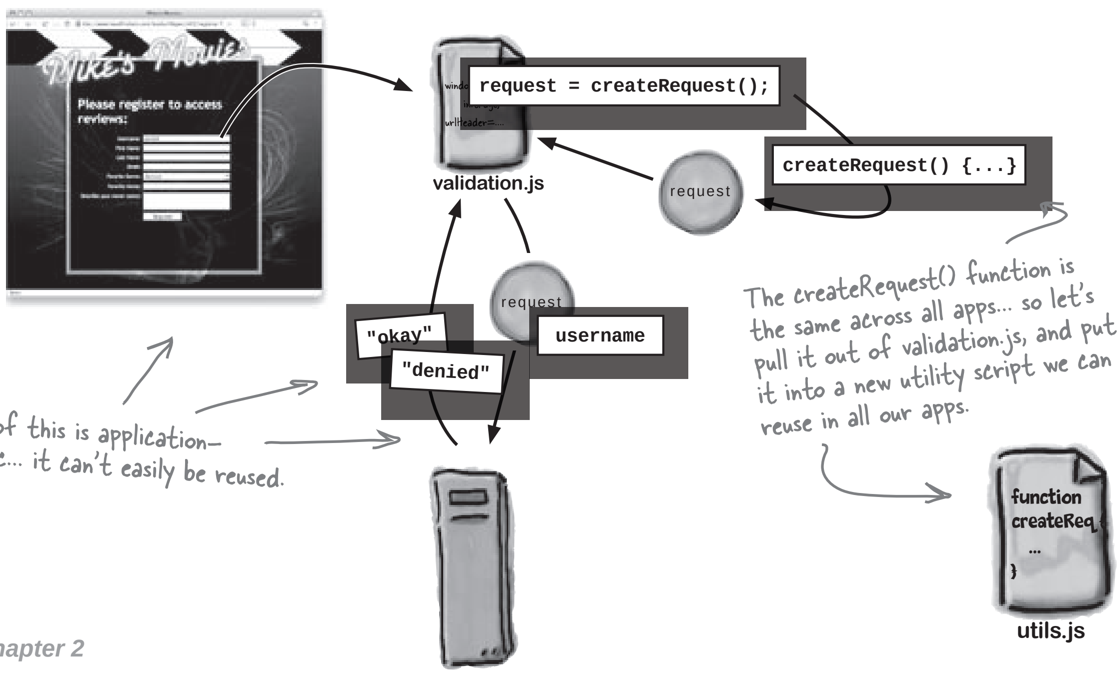


Copy and paste is not good code reuse.

The `createRequest()` function for Mike's movie site is identical to the `createRequest()` function from Rob's site in Chapter 1. And copying that code from the script you wrote in Chapter 1 into your new validation.js is a bad idea. If you need to make a change, you'll now have to make it in two places. And what do you think will happen when you've got ten or twenty Ajax apps floating around?

When you find code that's common across your apps, take that code out of application-specific scripts, and put it into a reusable utility script. So for `createRequest()`, we can pull it out of `validation.js` in the movie site and create a new script. Let's call it `utils.js` and start putting anything that's common to our apps into it.

Then, each new app we write can reference `utils.js`, as well as a script for application-specific JavaScript.



☐ Create a new file and name it `utils.js`. Add the `createRequest()` function from the last chapter, or from page 61, into the script, and save your changes.

Make each of these changes to your own code, and check off the boxes as you go.



utils.js

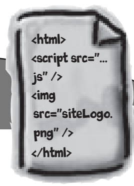
```
function createRequest() {
  try {
    request = new XMLHttpRequest();
  } catch (tryMS) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (otherMS) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = null;
      }
    }
  }

  return request;
}
```

☐ Open up `registration.html`, and add a new `<script>` tag referencing the new JavaScript, `utils.js`.

It's usually a good idea to put your utility code first and your application-specific code second. Getting into habits like this will give all your code a familiar, organized feel.

```
<head>
    <title>Mike's Movies</title>
    <link href="movies.css" rel="stylesheet" type="text/css" />
    <script src="scripts/utils.js" type="text/javascript"></script>
    <script src="scripts/validation.js" type="text/javascript"></script>
</head>
```



registration.html

☐ If you've already added `createRequest()` to `validation.js`, be sure to remove that function. `createRequest()` should only appear in your `utils.js` script now.

there are no Dumb Questions

Q: Why did you reference utils.js ahead of validation.js?

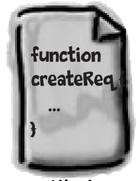
A: Lots of times your application-specific code will call your utilities. So it's best to make sure the browser parses your utility code before it parses any code that might call those utilities. Besides, it's a nice way to keep things organized: utilities first, application-specific code second.

Q: But I still don't understand how createRequest() actually works. What gives?

A: Good question. We've identified `createRequest()` as reusable and moved it into a utility script. That's a good thing, but we've still got to figure out what all that code is actually doing.

Separate what's the same across applications, and turn that code into a reusable set of functions.

Create a request object... on *multiple* browsers



utils.js

It's time to break into JavaScript and figure out exactly what's going on. Let's walk through exactly what each piece of `createRequest()` does, step by step.

1 Create the function

Start by building a function that any other code can call when it needs a request object.

```
function createRequest() {
  // create a variable named "request"
}
```

This function can be called from anywhere in our application.

No matter what syntax we use to get it, the request object will behave the same once we have an instance of it.

This insulates the calling code from all the messy details of browser compatibility.

2 Try to create an XMLHttpRequest for non-Microsoft browsers

Define a variable called request, and try to assign to it a new instance of the `XMLHttpRequest` object type. This will work on almost all browsers except Microsoft Internet Explorer.

```
function createRequest() {
  try {
    request = new XMLHttpRequest();
  } catch (tryMS) {
    // it didn't work, so we'll try something else
  }
}
```

XMLHttpRequest works on Safari, Firefox, Mozilla, Opera, and most other non-Microsoft browsers.

3 Try to create an ActiveXObject for Microsoft browsers

In the **catch** block, we try to create a request object using the syntax that's specific to Microsoft browsers. But there are two *different* versions of the Microsoft object libraries, so we'll have to try both of them.

```
try {
  request = new ActiveXObject("Msxml2.XMLHTTP");
} catch (otherMS) {
  try {

    request = new ActiveXObject("Microsoft.XMLHTTP");
  } catch (failed) {
    // that didn't work either--we just can't get a request object
  }
}
```

Most versions of IE support this syntax...

...but some of them require a different library.

All of this code here...

...goes in here.

4 If all else fails, return null

We've tried three different ways of obtaining a request object. If the parser reaches this request block, that means they've all failed. So declare `request` as null, and then let the calling code decide what to do about it. Remember, `null` is the object you have when you don't have an object.

```
request = null;
```

This goes in the final catch block.

Returning null puts the burden on the calling code, which can decide how to report an error.

5 Put it together, and return request

All that's left is to return `request`. If things went okay, `request` points to a request object. Otherwise, it points to `null`:

```
function createRequest() {
  try {
    request = new XMLHttpRequest();
  } catch (tryMS) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (otherMS) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = null;
      }
    }
  }
  return request;
}
```

For non-Microsoft browsers

For the Internet Explorer fans out there

We could generate an error here, but we'll let the calling code decide what to do if we can't get a request object.

No matter what, something's returned even if it's just a null value.

BULLET POINTS

- Different browsers use different syntax to obtain a request object. Your code should account for each type of syntax, so your app works in multiple browsers.
- No matter what syntax you use to get an instance of the request object, the object itself always behaves the same way.
- Returning a null if you can't get an instance of the request object lets the calling code decide what to do. That's more flexible than generating an error.

Ajax app design involves both the web page AND the server-side program

Even though there was already a web form for Mike's registration page, we've got to interact with that form to get the user's username, and later on, to update the page with an error message if the selected username's taken.

And even though we're letting someone else worry about writing the server-side code, we've still got to know ***what*** to send to that code... and ***how*** to send that information.

Take a look at the steps we need to perform to check a username for validity. Most of these steps are about interacting with either the web form or a server-side program:

This is what the call to createRequest() does.

1. **Try to get a request object**
2. **Show an alert if the browser can't create the request**
3. **Get the username the user typed into the form**
4. **Make sure the username doesn't contain problematic characters for an HTTP request**
5. **Append the username to server url**
6. **Tell the browser what function to call when the server responds to the request**
7. **Tell the browser how to send the request to the server**
8. **Send the request object**

Remember, createRequest() doesn't handle errors, so we'll need to do that ourselves.

This interacts with the web form.

These have to do with getting the username to the server.

This is the "callback." We'll write it in a few pages.

Now we're through until the request returns, and the browser gives it to the callback.

Here's more server interaction.

Good Ajax design is mostly about interactions. You've got to interact with your users via a web page, and your business logic via server-side programs.



Code Magnets

Most of the code for the checkUsername() function is scrambled up on the fridge. Can you reassemble it? The curly braces fell on the floor, and they were too small to pick up. Feel free to add as many of those as you need.

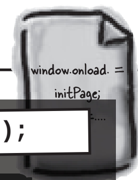
```
function checkUsername() {







}
```



validation.js

```
request.send(null);
alert("Unable to create request");
var theName = document.getElementById("username").value;
if (request == null)
} else {
request.open("GET", url, true);
request = createRequest();
request.onreadystatechange = showUsernameStatus;
var username = escape(theName);
var url = "checkName.php?username=" + username;
```



Code Magnet Solutions

Most of the code for the checkUserName() function is scrambled up on the fridge. Your job was to reassemble the code into a working function.

```
function checkUsername() {
  request = createRequest();
    if (request == null)
      alert("Unable to create request");
  } else {
    var theName = document.getElementById("username").value;
    var username = escape(theName);
    var url = "checkName.php?username=" + username;
    request.onreadystatechange = showUsernameStatus;
    request.open("GET", url, true);
    request.send(null);
  }
}
```

First, we call our utility function in utils.js to get the request object.



utils.js

If we get back a null, the function failed...

...so we'll tell the user.

getElementById grabs the element on the form with the id "username."

value is what the user actually typed in.

The JavaScript escape function cleans up what the user entered, just in case there are things like spaces or question marks in the text.

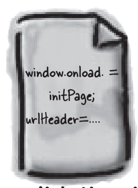
We're appending the username to the URL.

This is the callback that the browser will send the request object to when the server answers the request.

This tells the browser how to send the request. We're using the "GET" form method and sending it to the url contained in the url variable. And "true" means it's going asynchronously—the user can keep filling out the form while the server checks their username.

send() actually sends the request object off to the server. The null means we're not sending any other information along with it.

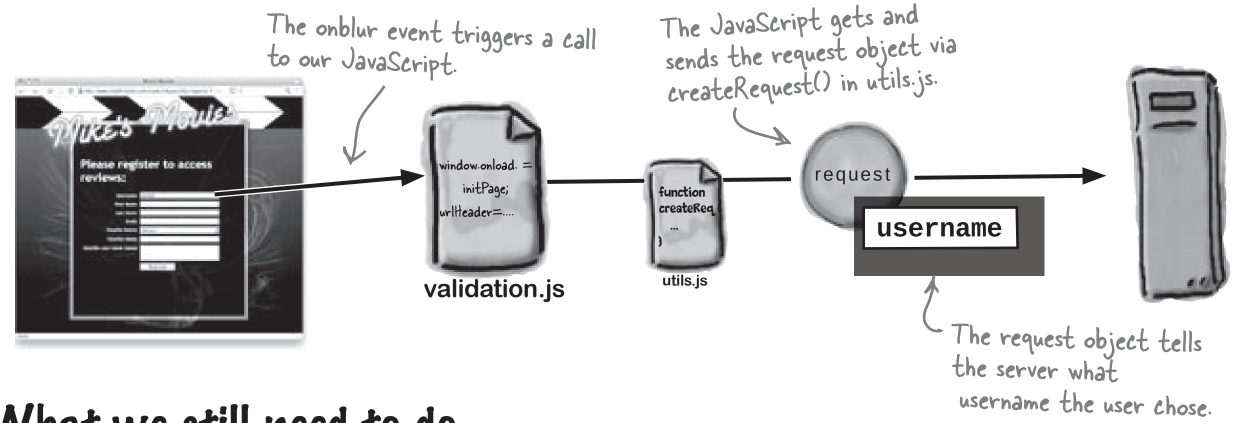
This code all belongs in validation.js.



validation.js

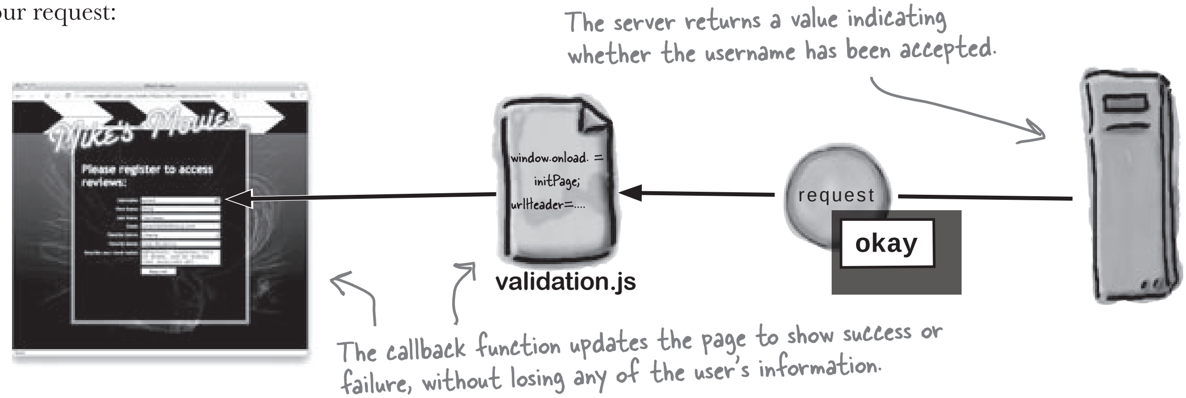
What we've done so far...

Now we've got everything ready to make a request to the server when a new username is entered in.



What we still need to do...

Now we're just about ready to actually have the server respond to our request:



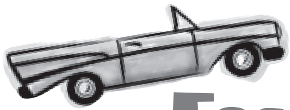
there are no Dumb Questions

Q: What does that getElementById() thing do exactly?

A: We'll talk about `getElementById()` a *lot* when we look at the DOM in Chapters 5 and 6. For right now, all you need to understand is that it returns a JavaScript object that represents an XHTML element on a web page.

Q: And "value"? What's that?

A: The `getElementById()` function returns a JavaScript object that represents an XHTML element. Like all JavaScript objects, the object the function returns has properties and methods. The `value` property contains the text that the element contains, in this case, whatever the user entered into the username field.



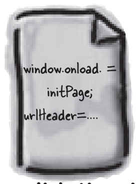
Test Drive

Let's make sure everything's working before moving on...

The JavaScript still doesn't update the page in any way, but we can use a few more alerts to check that our `checkUsername()` function's working the way we want.

Open `validation.js` in your editor, and add the code inside the `checkUserName()` function that's shown below. It's the same as the magnet exercise you just did, but there are a few more alerts added to help track what the browser's doing.

Once you've entered the code, save the file, and load the page in your browser. Enter anything you'd like in the username field, and you should see all these alerts displayed.

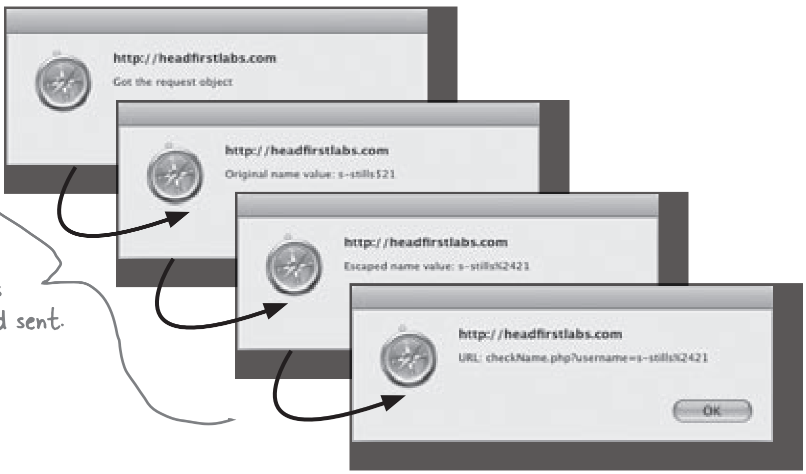


validation.js

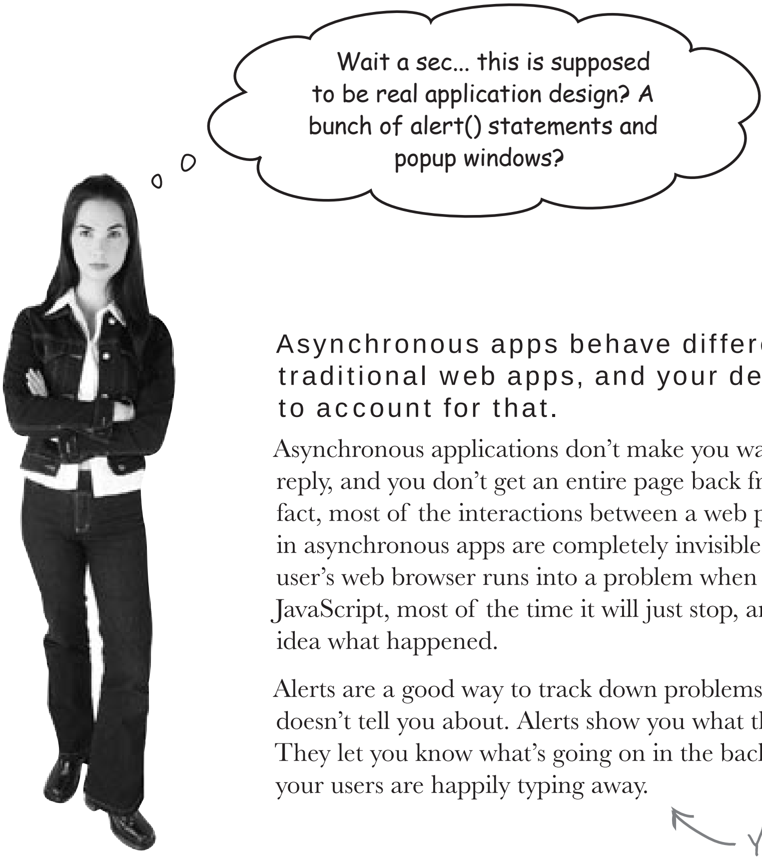
```
function checkUsername() {
  request = createRequest();
  if (request = null)
    alert("Unable to create request");
  else
  {
    alert("Got the request object");
    var theName = document.getElementById("username").value;
    alert("Original name value: " + theName);
    var username = escape(theName);
    alert("Escaped name value: " + username);
    var url = "checkName.php?username=" + username;
    alert("URL: " + url);

    request.onreadystatechange = userNameChecked;
    request.open("GET", url, true);
    request.send(null);
  }
}
```

These alerts are like status messages or debugging information... they let us know what's going on behind the scenes.



You should see an alert indicating the request is created, configured, and sent.



Asynchronous apps behave differently than traditional web apps, and your debugging has to account for that.

Asynchronous applications don't make you wait for a server's reply, and you don't get an entire page back from the server. In fact, most of the interactions between a web page and a server in asynchronous apps are completely invisible to a user. If the user's web browser runs into a problem when it executes some JavaScript, most of the time it will just stop, and you'll have no idea what happened.

Alerts are a good way to track down problems the browser doesn't tell you about. Alerts show you what the *browser* sees. They let you know what's going on in the background while your users are happily typing away.

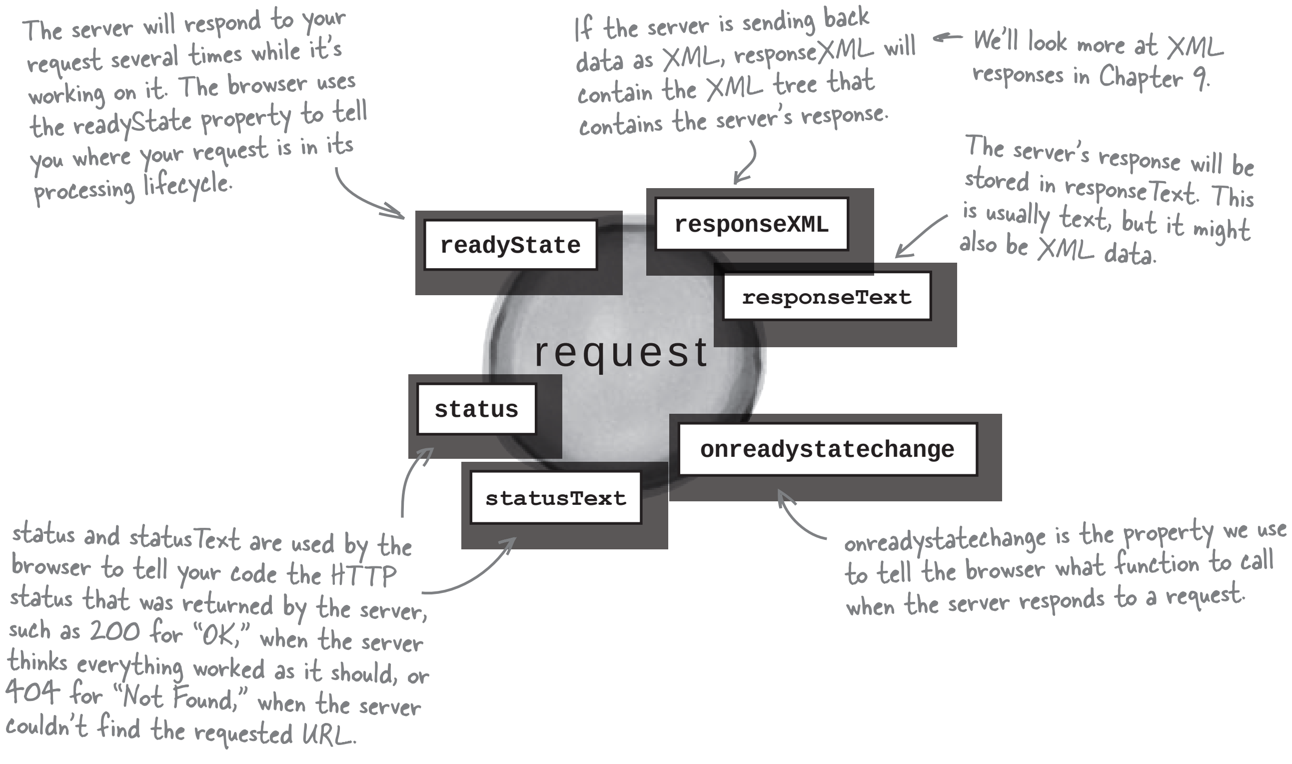
You'll want to take out all these alerts once you've tracked down any problems.

You can't usually rely on a server to tell you there's a problem in asynchronous apps. It's YOUR job to figure out if there's a problem, and respond to it in a useful manner.

The request object connects your code to the web browser

All we have left to do is write the code that the browser will call when the server responds to the request. That's where the request object comes into play. It lets us tell the browser what to do, and we can use it to ask the browser to make a request to the server and give us the result.

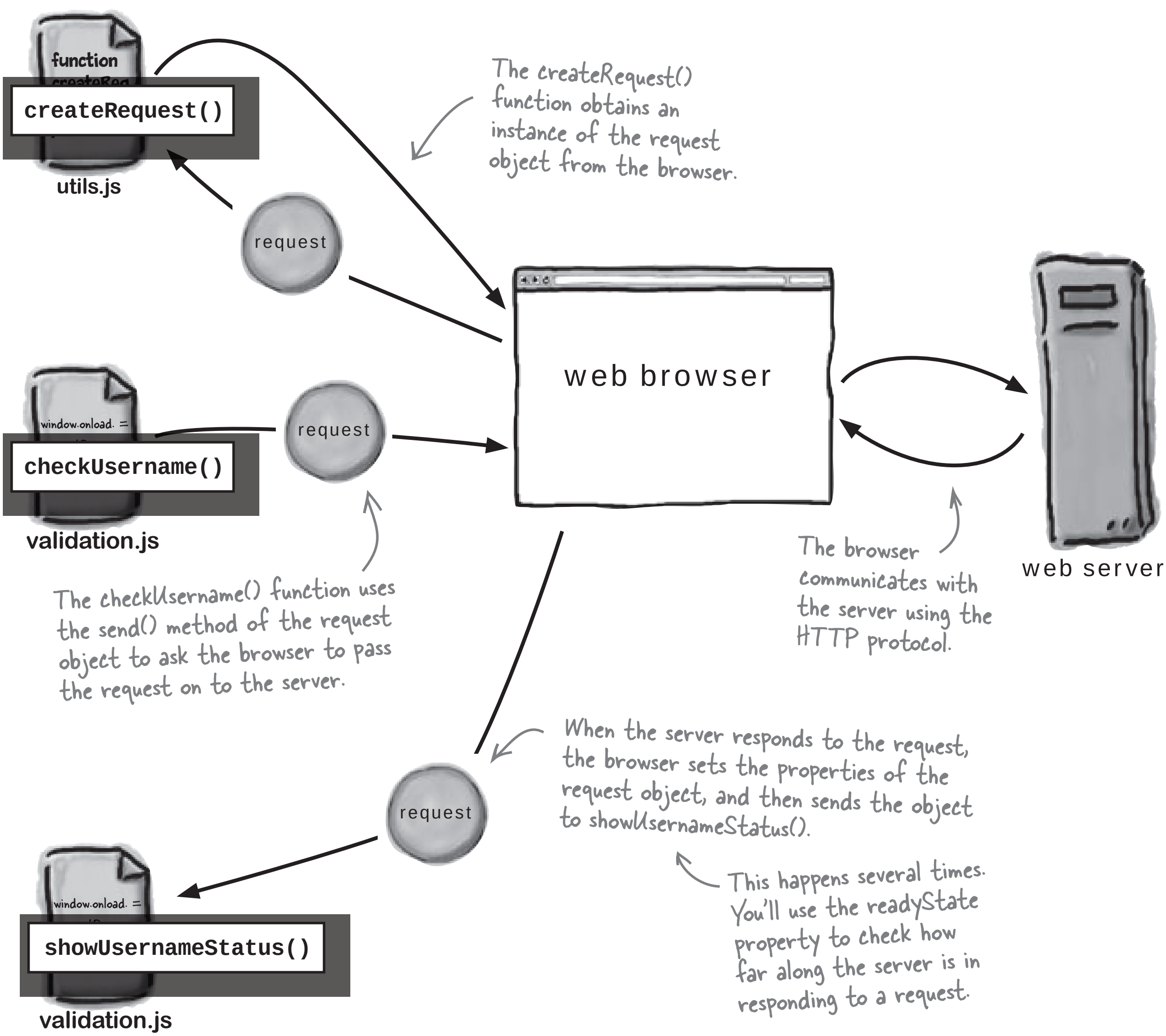
But how does that actually happen? Remember, ***the request object is just an ordinary JavaScript object***. So it can have properties, and those properties can have values. There are several that are pretty useful. Which do you think we'll need in our callback function?



The browser makes the server's response available to your code through the properties of the request object.

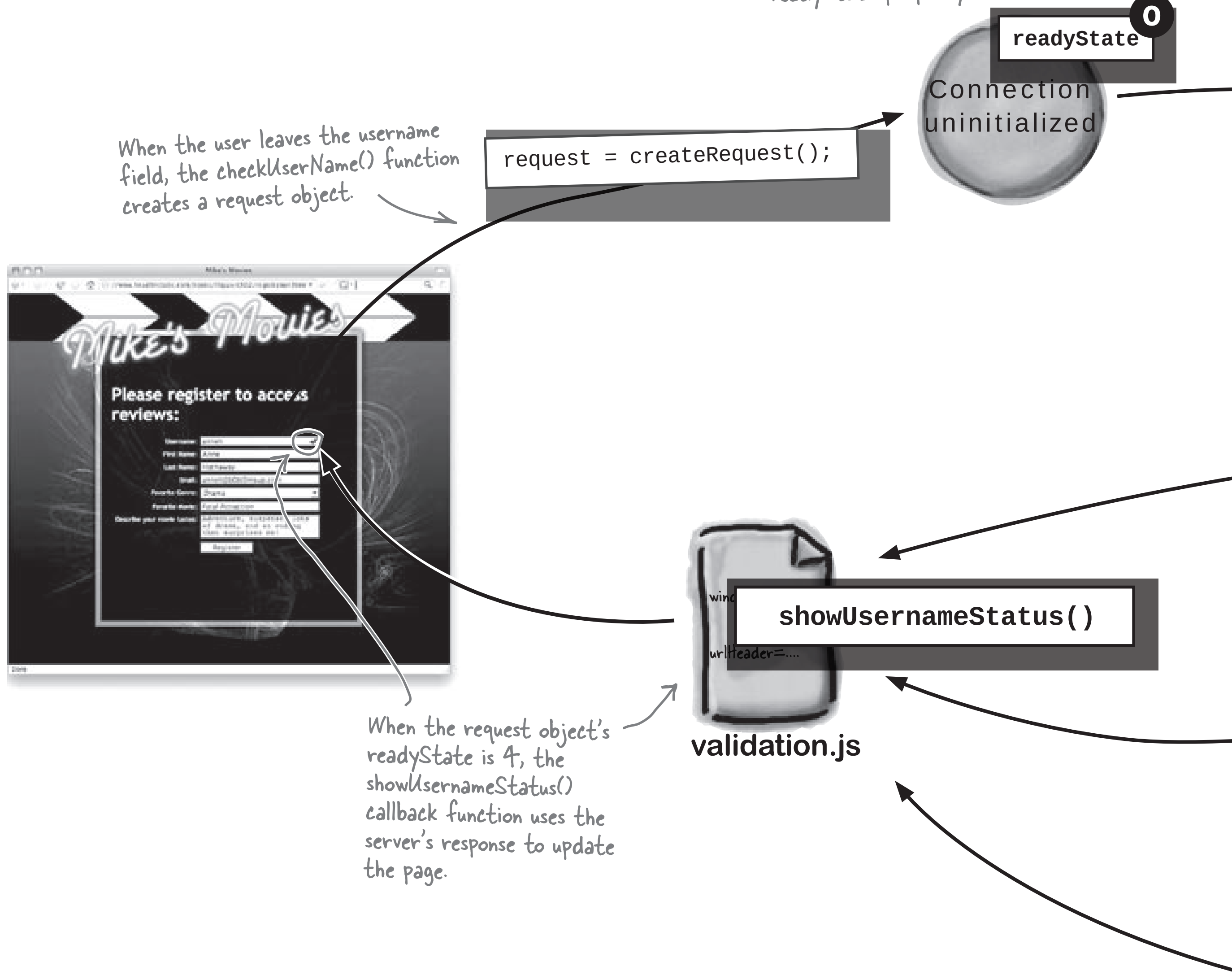
You talk to the browser, not the server

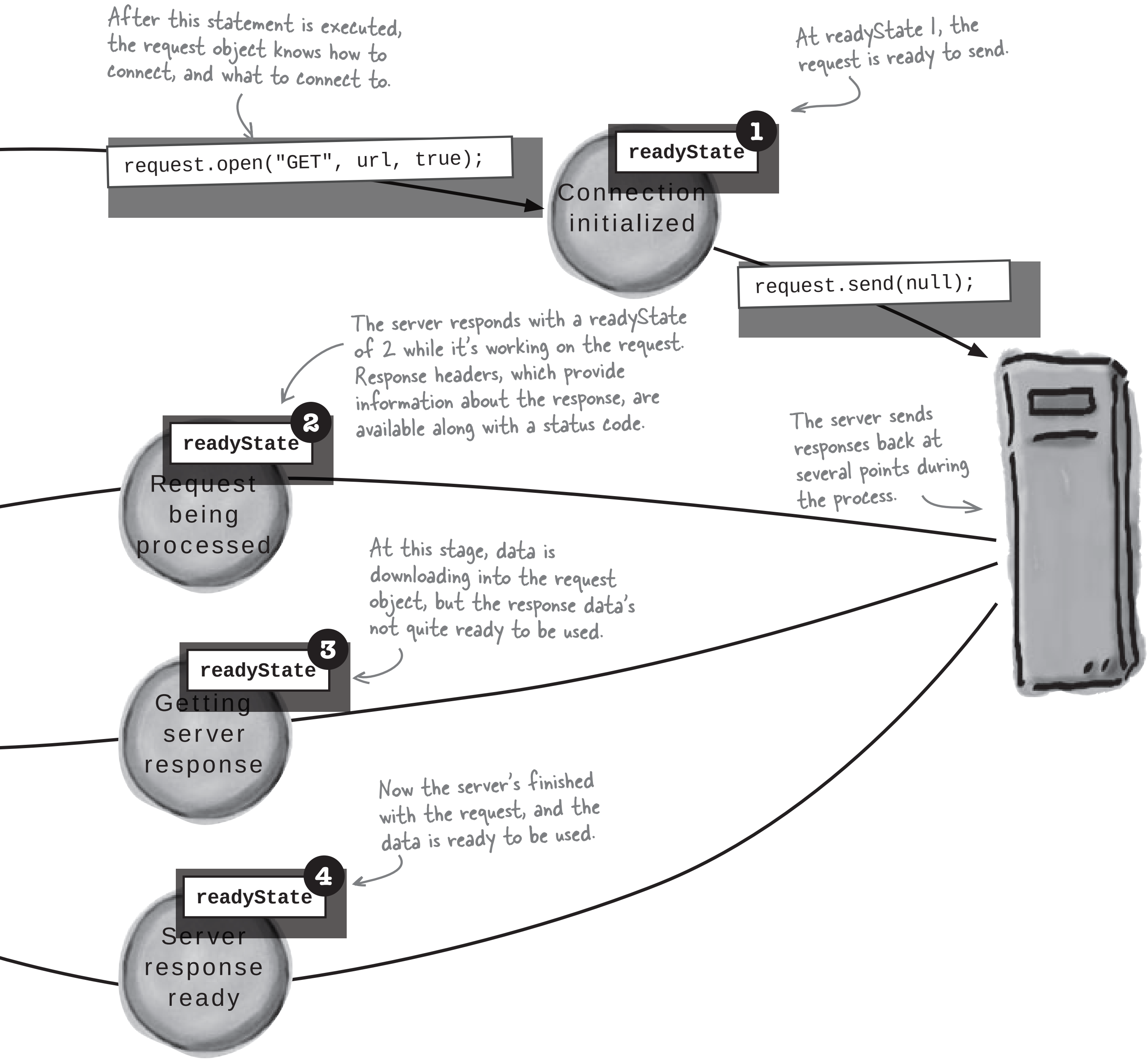
Although it's easy to talk about your code "sending a request object to the server," that's not exactly what happens. In fact, you talk to the web browser, not the server, and the browser talks to the server. **The browser sends your request object to the server,** and **the browser translates the server's response** before giving that response data back to your web page.



Ready states up close

The browser uses the `readystate` property of the request object to tell your callback function where a request is in its lifecycle. Let's take a look at exactly what that means.



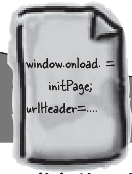

After this statement is executed, the request object knows how to connect, and what to connect to.
request.open("GET", url, true);
At readyState 1, the request is ready to send.
readyState 1
Connection initialized
request.send(null);
The server responds with a readyState of 2 while it's working on the request. Response headers, which provide information about the response, are available along with a status code.
readyState 2
Request being processed
The server sends responses back at several points during the process.
At this stage, data is downloading into the request object, but the response data's not quite ready to be used.
readyState 3
Getting server response
Now the server's finished with the request, and the data is ready to be used.
readyState 4
Server response ready

The browser calls back your function with the server's response

Every time the response object's `readyState` property changes, the browser has to do something. And what does it do? It runs the function assigned to the request object's `onreadystatechange` property:

Every time the ready state of the response changes – which is every time the server updates the browser on the request its processing – the browser calls this function.

```
function checkUsername() {
  request = createRequest();
  ...
  request.onreadystatechange = showUsernameStatus;
  ...
}
```



validation.js

In your callback function, you need to make sure that the response is actually ready for you to use. You can check the `readyState` property and the server status, and then take action based on the server's response:

This is the function name we used for the onreadystatechange property. If the name doesn't match exactly, the function won't be called.

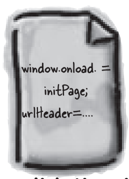
```
function showUsernameStatus() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      if (request.responseText == "okay") {
        // if it's okay, no error message to show
      }
      else {
        // if there's a problem, we'll tell the user
        alert("Sorry, that username is taken.");
      }
    }
  }
}
```

This if statement makes sure that none of the rest of the code runs unless the readystate is "4," meaning the server is finished.

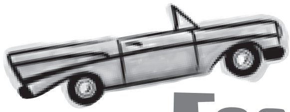
The server sends a status of "200" if everything is okay.

responseText is the text value the server sends back to us. If it's "okay," the username is free.

This code goes in validation.js, too.



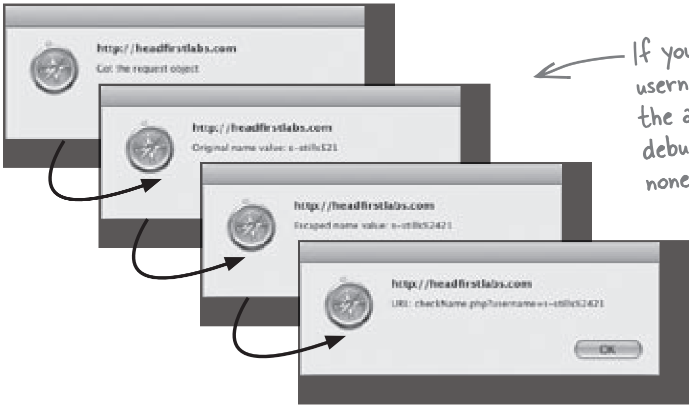
validation.js



Test Drive

Add the showUsernameStatus() function to validation.js, and load the registration page in your browser.

Try entering any username except "bill" or "ted." Your browser should display all the alerts we added to test the `initPage()` and `checkUsername()` functions.



If you enter a valid username, you'll get the alerts from your debugging code, but none indicate an error.

Now try entering "bill" or "ted" as the username. You should get the error message that's displayed by `showUsernameStatus()`.



This message should be displayed if you enter "bill" or "ted," and then leave the username field. Someone with that username is already registered.

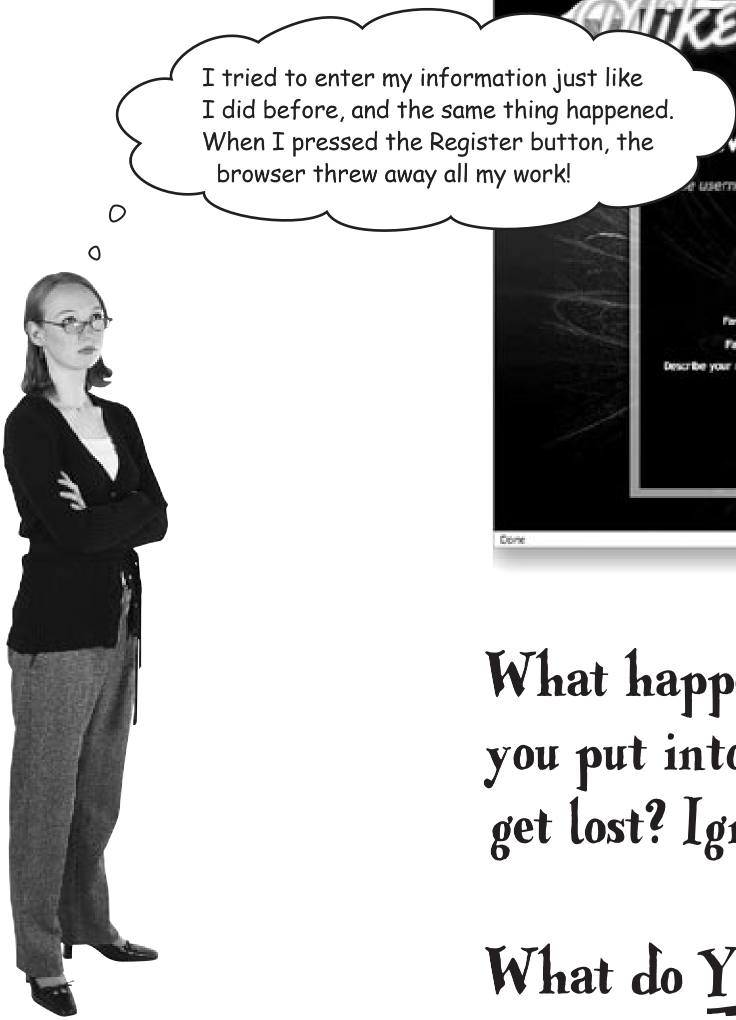
Once you're sure everything's working, go ahead and remove all those alert statements in `checkUsername()` that you added to test the code. The only alerts that should be left are to report that a request can't be created, in `checkUsername()`, and to report a username's already taken, in `showUsernameStatus()`.

Now that you're sure the interaction between your code and the server works, you don't need those alert() debugging statements anymore.

Show the Ajax registration page to Mike...

Everything works. But when you give all your code to Mike, and he goes live with the new improved registration page, there are still some problems:





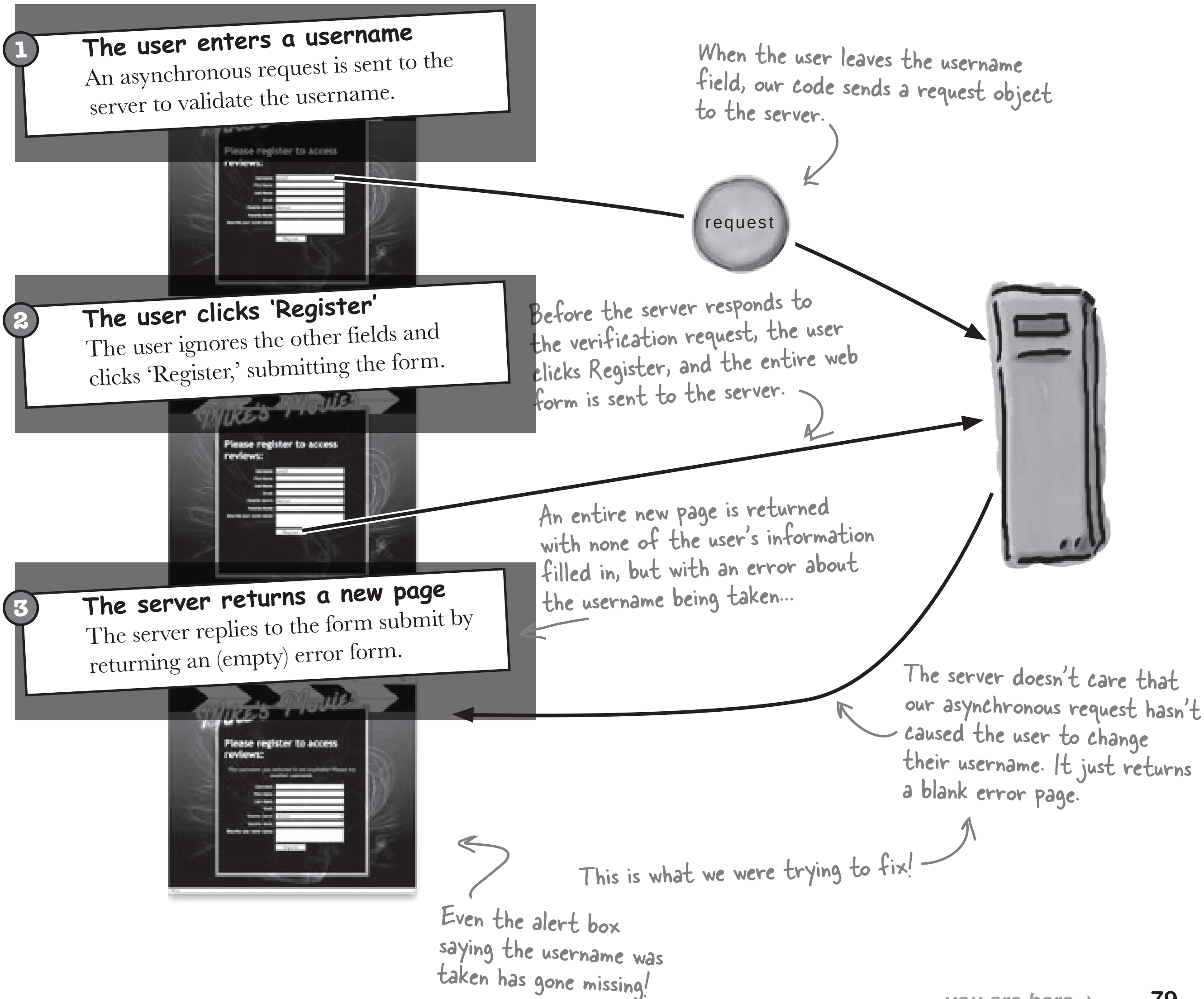
What happened? Did all the work you put into the registration page get lost? Ignored?

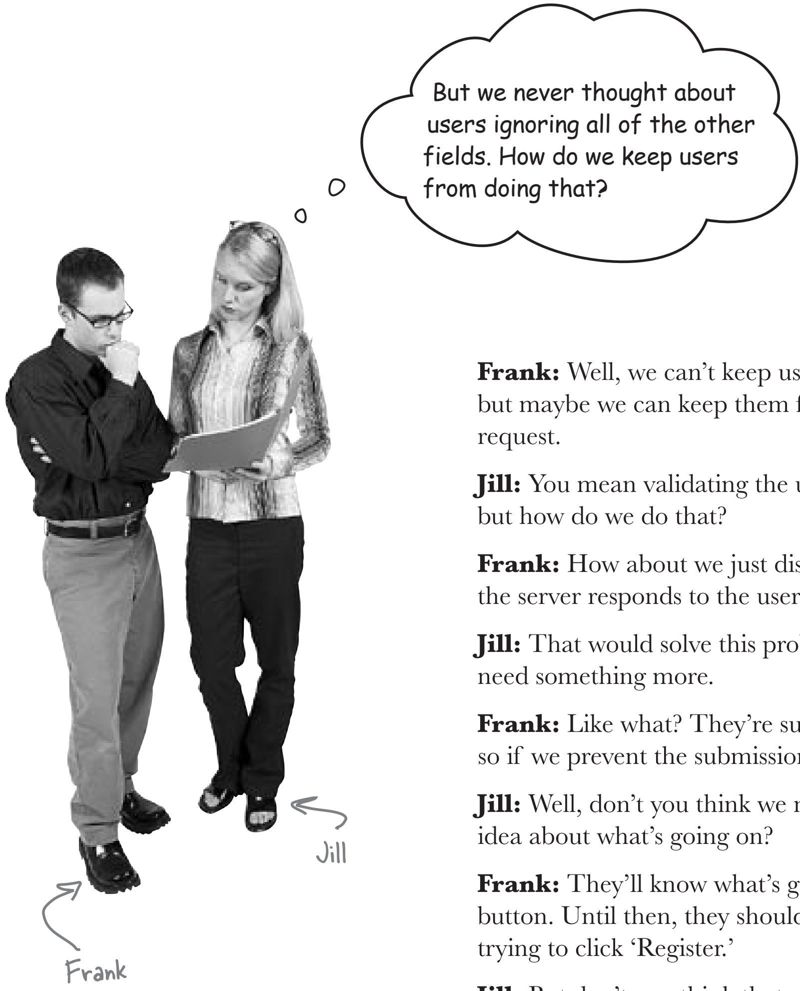
What do YOU think?

The web form has TWO ways to send requests to the server now

Suppose a user does just what you expect: they enter a username, and while an asynchronous request is going to the server and getting handled by the browser, your callback is running, and the user's filling out other information on the form. Everything works great—just like you planned.

But suppose the user's so eager to get to Mike's review of Iron Man that they put in their username, ignore everything else on the form, and click "Register." What happens then?





Frank: Well, we can't keep users from skipping over fields, but maybe we can keep them from getting ahead of our request.

Jill: You mean validating the username? Yeah, that's perfect, but how do we do that?

Frank: How about we just disable the Register button until the server responds to the username validation request.

Jill: That would solve this problem, but it seems like we need something more.

Frank: Like what? They're submitting the form too soon, so if we prevent the submission, the problem's solved.

Jill: Well, don't you think we need to give the user some idea about what's going on?

Frank: They'll know what's going on when we enable the button. Until then, they should be filling out the form, not trying to click 'Register.'

Jill: But don't you think that might be confusing? If the user finishes filling out the form, or doesn't want to fill it all out, then they're just going to be sitting there, stuck, and they won't know why.

Frank: Well, we need to let them know the application is doing something. What about displaying a message?

Jill: Another alert? That's just going to annoy them in a different way. How about a graphic? We could display an image when we send the request to the browser...

Frank: ...and another when their username's verified.

Jill: Hey, and if we used an image to show whether the username is okay or not, we could get rid of the alert when there's a problem with the username, too.

Frank: Perfect! Visual feedback ***without*** annoying popups. I love it!

You can never assume your users will do things exactly the way you do... plan for EVERYTHING!

☐ **Display an "In Progress" graphic during verification requests**

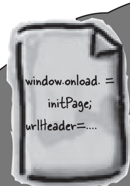
When we send a request to the server to verify a username, we'll display a graphic next to the username field, telling the user what's going on. That way, they'll know exactly what's happening as they work through the form.

Make each of these changes to your own code, and check off the boxes as you go.

getElementById is probably starting to look familiar. It lets you access an element on an XHTML page.

```
function checkUsername() {
  document.getElementById("status").src = "images/inProcess.png";
  request = createRequest();
  ...
}
```

Displaying this image tells the user something's going on.



validation.js

☐ **Display a status message upon verification**

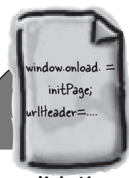
Once the request object returns, we can display another graphic in our callback function. If the username is okay, the graphic indicates that; otherwise, we'll show an error icon.

```
function showUsernameStatus() {
  ...
      if (request.responseText == "okay") {
        document.getElementById("status").src = "images/okay.png";
      }
      else {
        alert("Sorry, that user name is taken.");
        document.getElementById("status").src = "images/inUse.png";
        ...
      }
  ...
}
```

This graphic is displayed if the server says the username is okay.

We can ditch the alert popup in favor of a nicer graphical icon. (The `alert(...)` line is crossed out.)

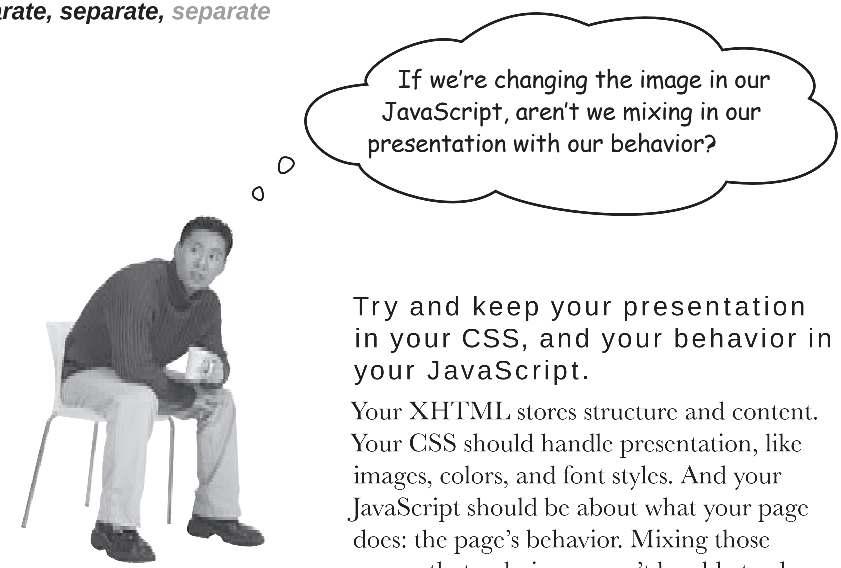
This graphic is shown if the username is taken.



validation.js

BRAIN POWER

What do you think about this approach? Does it follow the principle of separating content from presentation? Would you change anything?



Try and keep your presentation in your CSS, and your behavior in your JavaScript.

Your XHTML stores structure and content. Your CSS should handle presentation, like images, colors, and font styles. And your JavaScript should be about what your page does: the page's behavior. Mixing those means that a designer won't be able to change an image because it's in your code. Or a programmer will have to mess with a page author's structure. That's never a good thing.

It's not always possible, but when you can, keep your presentation in your CSS, and use JavaScript to interact with the CSS rather than affecting the presentation of a page directly.

Let's create CSS classes for each state of the processing...

Instead of changing an image directly, let's put all the image details in our CSS. Open up `movies.css` and add the following CSS selectors:

This first class just sets up the location for the process icons...

...and these other three classes change out the image in that location.

```
... existing CSS ...
#username { padding: 0 20px 0 2px; width: 198px; }
#username.thinking { background: url("../images/inProcess.png"); }
#username.approved { background: url("../images/okay.png"); }
#username.denied { background: url("../images/inUse.png"); }
```

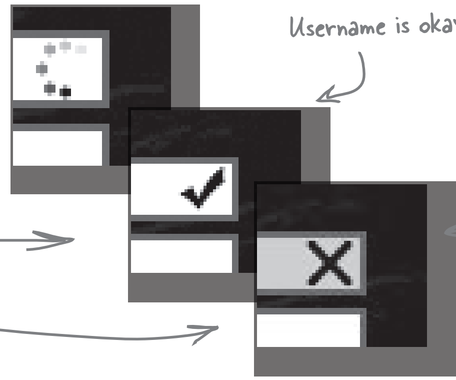


Add these four lines to your CSS.

These are the same images that we used in our JavaScript, but now they're in the CSS with the rest of the presentation.

...and change the CSS class with our JavaScript

Now our JavaScript doesn't need to know any image names, paths, or anything about ***how*** the process icons are being shown. Instead, we just need to know the three CSS classes that represent each stage of processing.

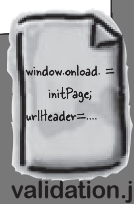


#username.thinking
#username.approved
#username.denied

Here are the three CSS class names.

Now we can update our JavaScript (again). This time we'll just change the CSS class instead of directly changing an image:

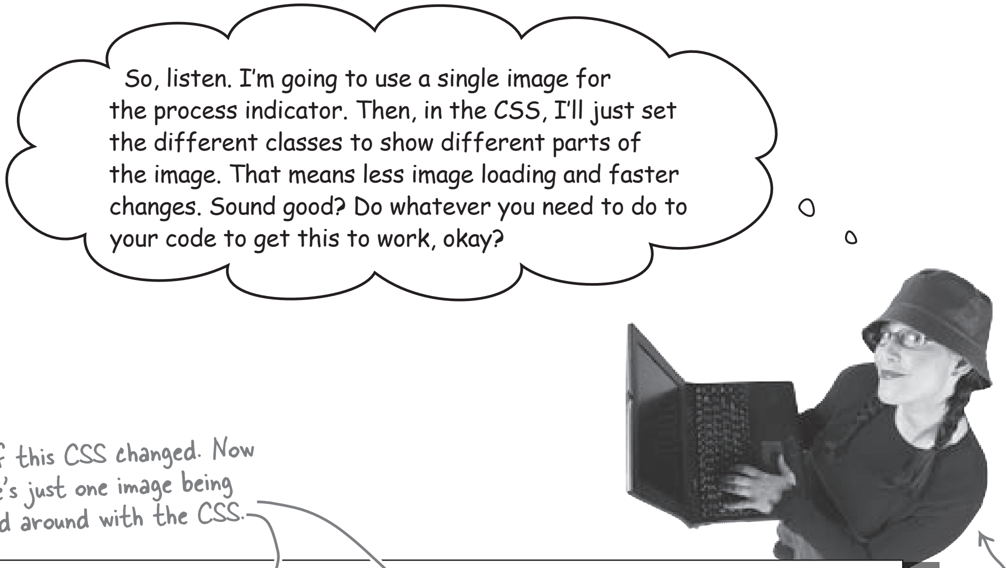
```
function checkUsername() {
  document.getElementById("status").src = "images/inProcess.png";
  document.getElementById("username").className = "thinking";
  request = createRequest();
  ...
```



You can change the CSS class using the className property of an element.

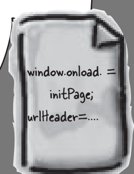
```
function showUsernameStatus() {
  ...
      if (request.responseText == "okay") {
        document.getElementById("status").src = "images/okay.png";
        document.getElementById("username").className = "approved";
      }
      else {
        alert("Sorry, that user name is taken.");
        document.getElementById("status").src = "images/inUse.png";
        document.getElementById("username").className = "denied";
      }
  ...
}
```

Remember to remove the lines that changed the image directly.



All of this CSS changed. Now there's just one image being moved around with the CSS.

```
... existing CSS ...
#username {
  background: #fff url('../images/status.gif') 202px 0 no-repeat;
  padding: 0 20px 0 2px; width: 198px; }
#username.thinking { background-position: 202px -19px; }
#username.approved { background-position: 202px -35px; }
#username.denied { background-color: #FF8282;
  background-position: 202px -52px; }
```



validation.js

Changes? We don't need no stinkin' changes!

Mike's web designer made lots of changes... but she didn't change the names of the CSS classes for each stage of processing. That means that ***all your JavaScript still works,*** with no updates! When you separate your content from your presentation, and separate both from your behavior, your web application gets **a *lot*** easier to change.

In fact, the CSS can change anytime, and we don't even need to know about it. As long as those CSS class names stay the same, our code will happily keep on working.

Good separation of content, presentation, and behavior makes your application a lot more flexible.

Only allow registration when it's appropriate

With process indicators in place, all that's left is to disable the Register button when the page loads, and then enable the button once a username's okay.

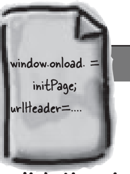
That involves just a few more changes to `validation.js`:

☐ **Disable the Register button**

When a user first loads the page, the username hasn't been checked. So we can disable the Register button right away in our initialization code.

By setting the disabled property to true, the user can fill in the fields, but they can't press the submit button until we're ready.

```
function initPage(){
  document.getElementById("username").onblur = checkUsername;
  document.getElementById("register").disabled = true;
}
```



validation.js

☐ **Enable the Register button**

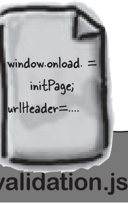
If the username is okay, the user's ready to register, so we need to enable the Register button. But if there's a problem with the username, they need to try again, so we should keep the Register button disabled. And just to make things easier for the user, let's move them back to the username field if their username is rejected:

```
function showUsernameStatus() {
  ...
      if (request.responseText == "okay") {
        document.getElementById("username").className = "approved";
        document.getElementById("register").disabled = false;
      }
      else {
        document.getElementById("username").className = "denied";
        document.getElementById("username").focus();
        document.getElementById("username").select();
        document.getElementById("register").disabled = true;
        ...
      }
  ...
}
```

If the username is okay, enable the Register button.

This moves the user back to the username field.

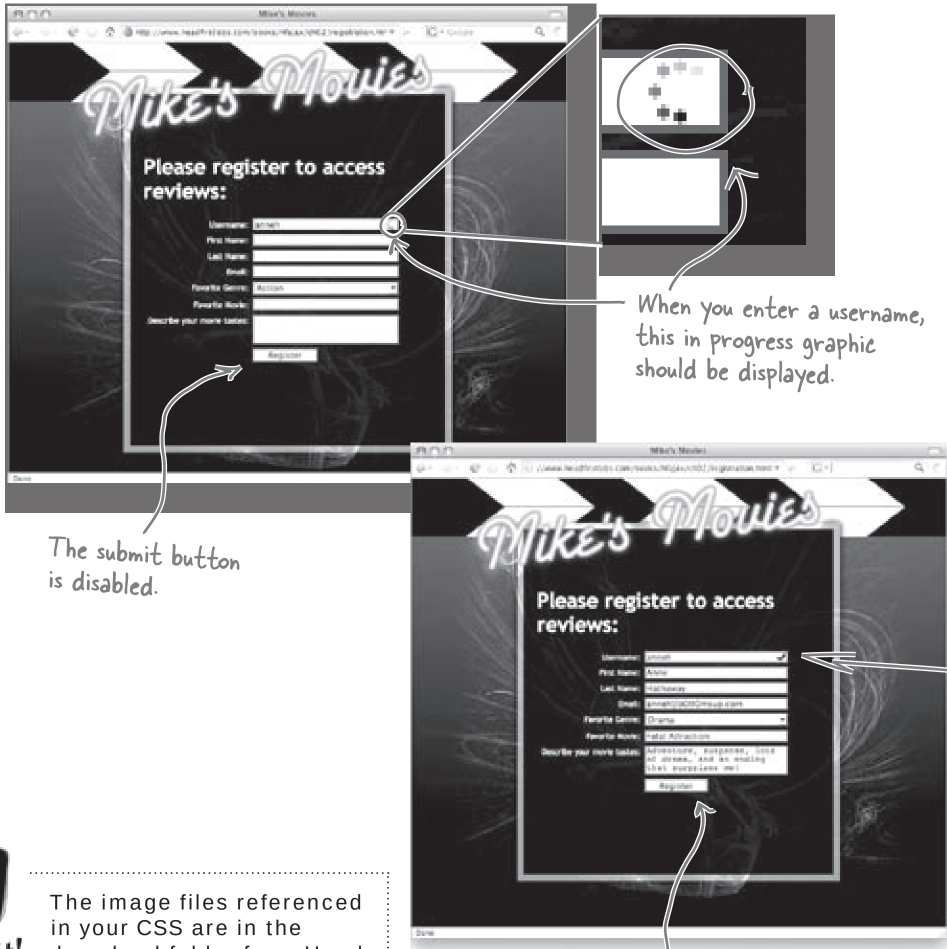
If the username's taken, make sure the Register button stays disabled.



validation.js

Test Drive

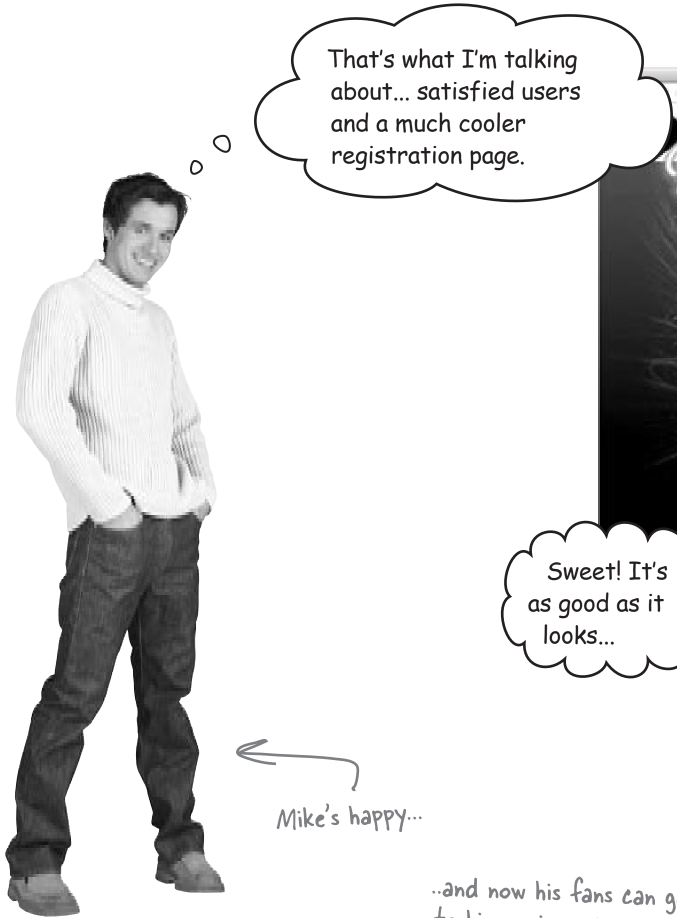
Make sure you've updated validation.js and mpovies.css, and load up Mike's registration page. Try it out to make sure everything's behaving like it should.



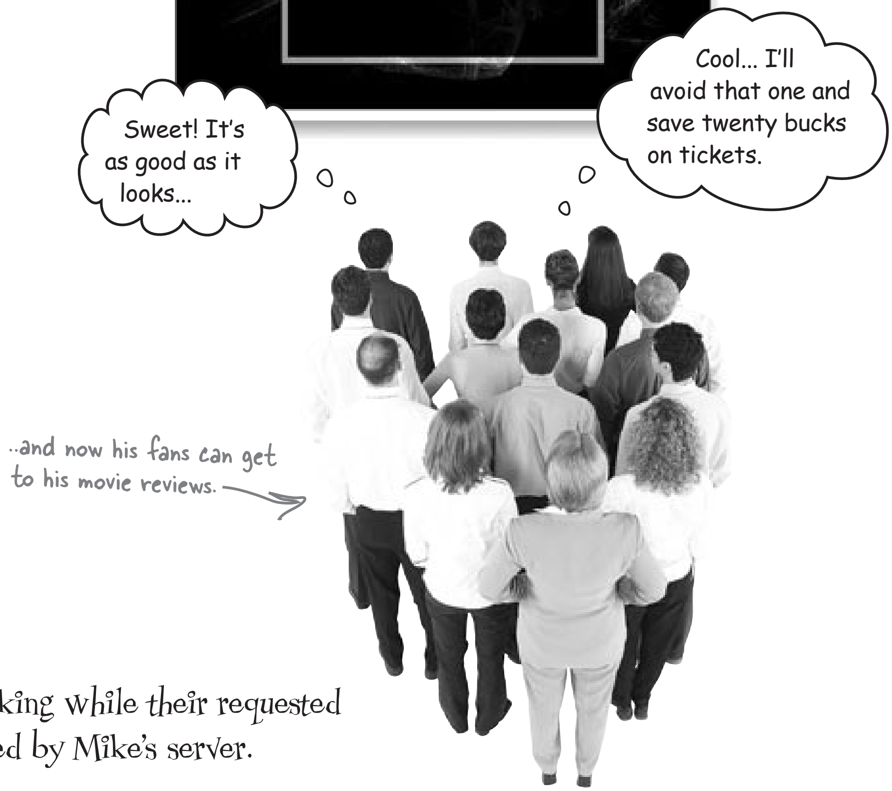
Watch it!

The image files referenced in your CSS are in the download folder from Head First Labs.

Be sure you've got the complete examples folder from Head First Labs, including the process indicator image.



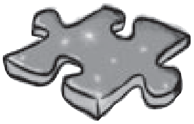




Now Mike's page...

- ...lets users keep working while their requested usernames are verified by Mike's server.
- ...prevents user mistakes by disabling buttons that aren't safe or appropriate to use, and enables those buttons when they <u>are</u> useful.
- ...doesn't annoy his users with intrusive popups, but still gives them useful visual feedback.

Along the way you started thinking about application design in an entirely new way... going beyond a traditional request/wait/response model.



Word Search

Take some time to sit back and give your right brain something to do. It's your standard word search; all of the solution words are from this chapter.

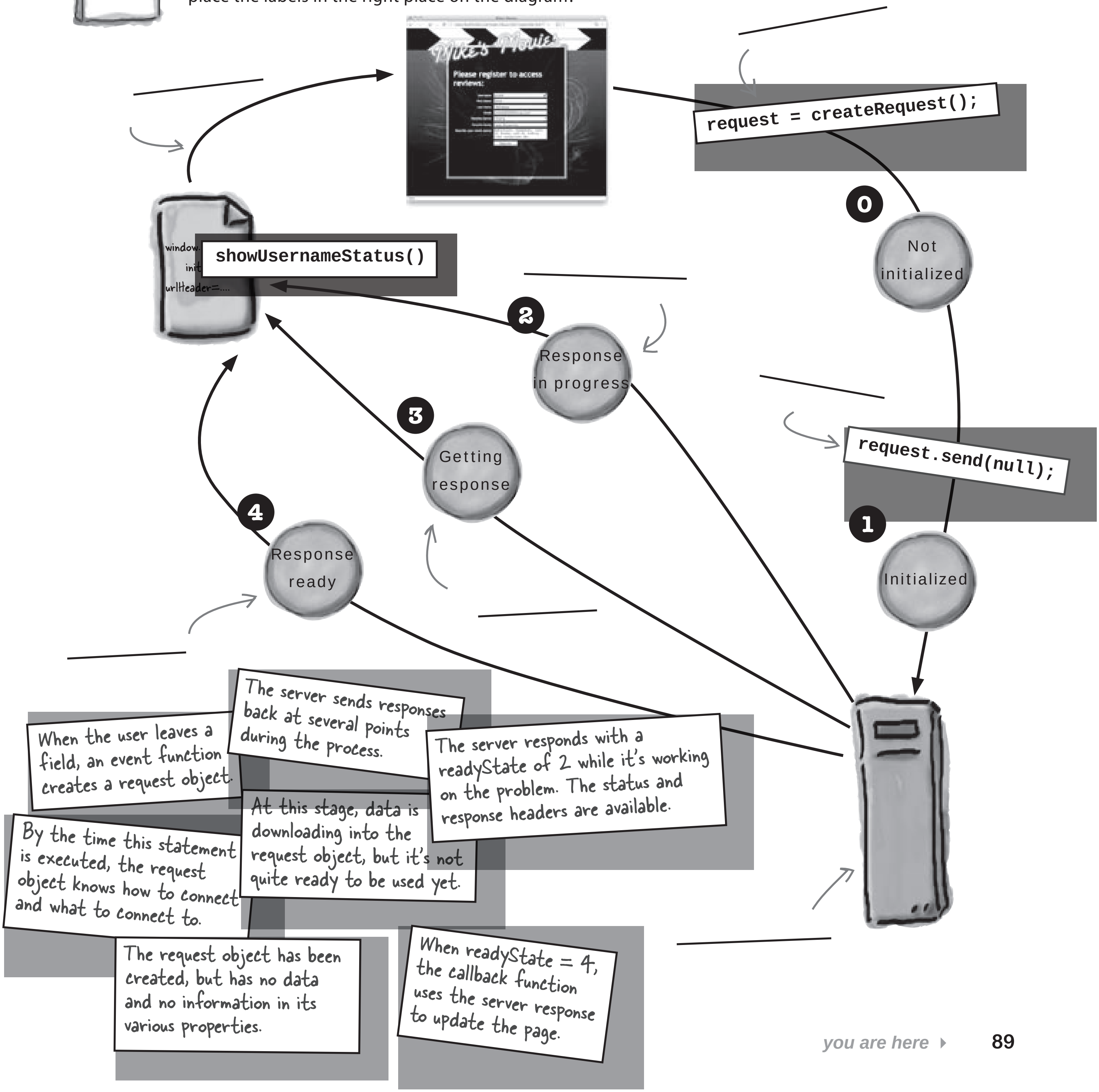
X	A	R	S	M	O	K	E	J	U	D	H	E
A	C	T	I	V	E	X	O	B	J	E	C	T
A	V	I	O	R	S	M	A	L	T	R	S	V
Q	S	L	H	O	C	L	V	J	A	R	S	L
J	U	Y	O	R	U	H	A	E	A	H	A	R
A	M	N	N	O	N	T	L	Y	H	E	R	A
Z	O	E	U	C	S	T	F	I	D	N	E	S
H	P	T	K	A	H	P	I	N	L	O	L	N
G	E	Y	C	C	E	R	L	O	X	L	B	R
A	N	I	A	H	R	E	O	A	U	D	G	R
O	U	N	B	E	D	Q	B	N	R	E	A	K
I	N	G	L	F	A	U	R	L	O	N	S	A
N	D	C	L	R	I	E	F	R	I	U	D	Y
A	R	E	A	D	Y	S	T	A	T	E	S	D
J	E	R	C	I	C	T	H	R	I	Z	A	R

Word list:

ActiveXObject
Asynchronous
Ajax
Cache
Callback
Null
Open
Readystate
Send
URL
XMLHttpRequest

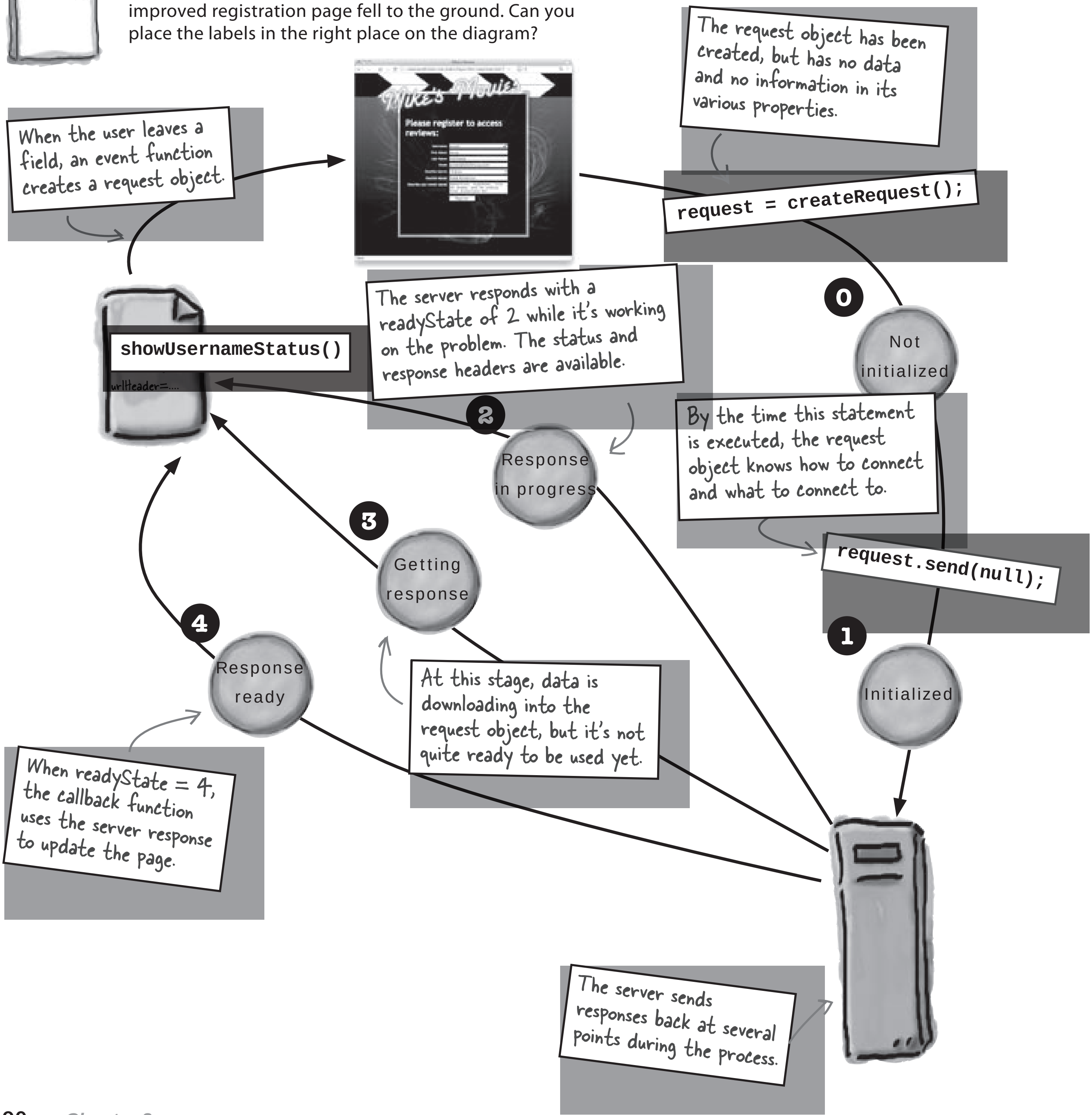
Label Magnets

All the labels describing what's going on in the new-and-improved registration page fell to the ground. Can you place the labels in the right place on the diagram?



Label Magnets Solution

All the labels describing what's going on in the new-and-improved registration page fell to the ground. Can you place the labels in the right place on the diagram?



Word Search Solution

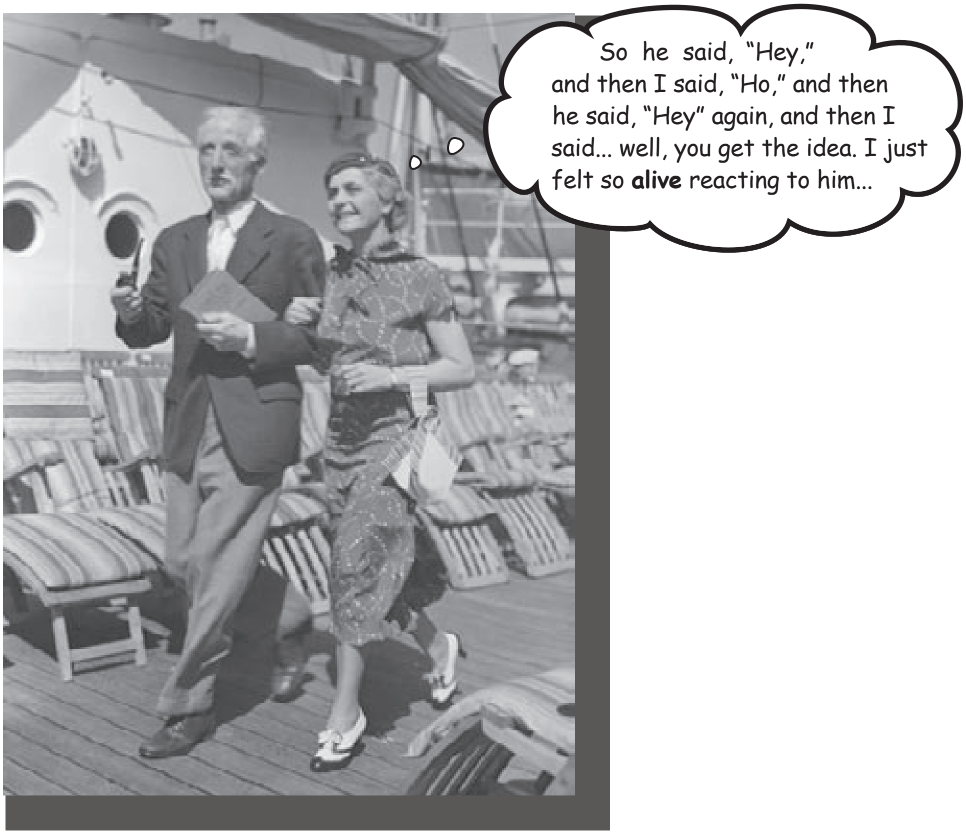
X	A	R	S	M	O	K	E	J	U	D	H	E
A	C	T	I	V	E	X	O	B	J	E	C	T
A	V	I	O	R	S	M	A	L	T	R	S	V
Q	S	L	H	O	C	L	V	J	A	R	S	L
J	U	Y	O	R	U	H	A	E	A	H	A	R
A	M	N	N	O	N	T	L	Y	H	E	R	A
Z	O	E	U	C	S	T	F	I	D	N	E	S
H	P	T	K	A	H	P	I	N	L	O	L	N
G	E	Y	C	C	E	R	L	O	X	L	B	R
A	N	I	A	H	R	E	O	A	U	D	G	R
O	U	N	B	E	D	Q	B	N	R	E	A	K
I	N	G	L	F	A	U	R	L	O	N	S	A
N	D	C	L	R	I	E	F	R	I	U	D	Y
A	R	E	A	D	Y	S	T	A	T	E	S	D
J	E	R	C	I	C	T	H	R	I	Z	A	R

Word list:

~~ActiveXObject~~
~~Asynchronous~~
~~Ajax~~
~~Cache~~
~~Callback~~
~~Null~~
~~Open~~
~~Readystate~~
~~Send~~
~~URL~~
~~XMLHttpRequest~~

3 javascript events

Reacting to your users

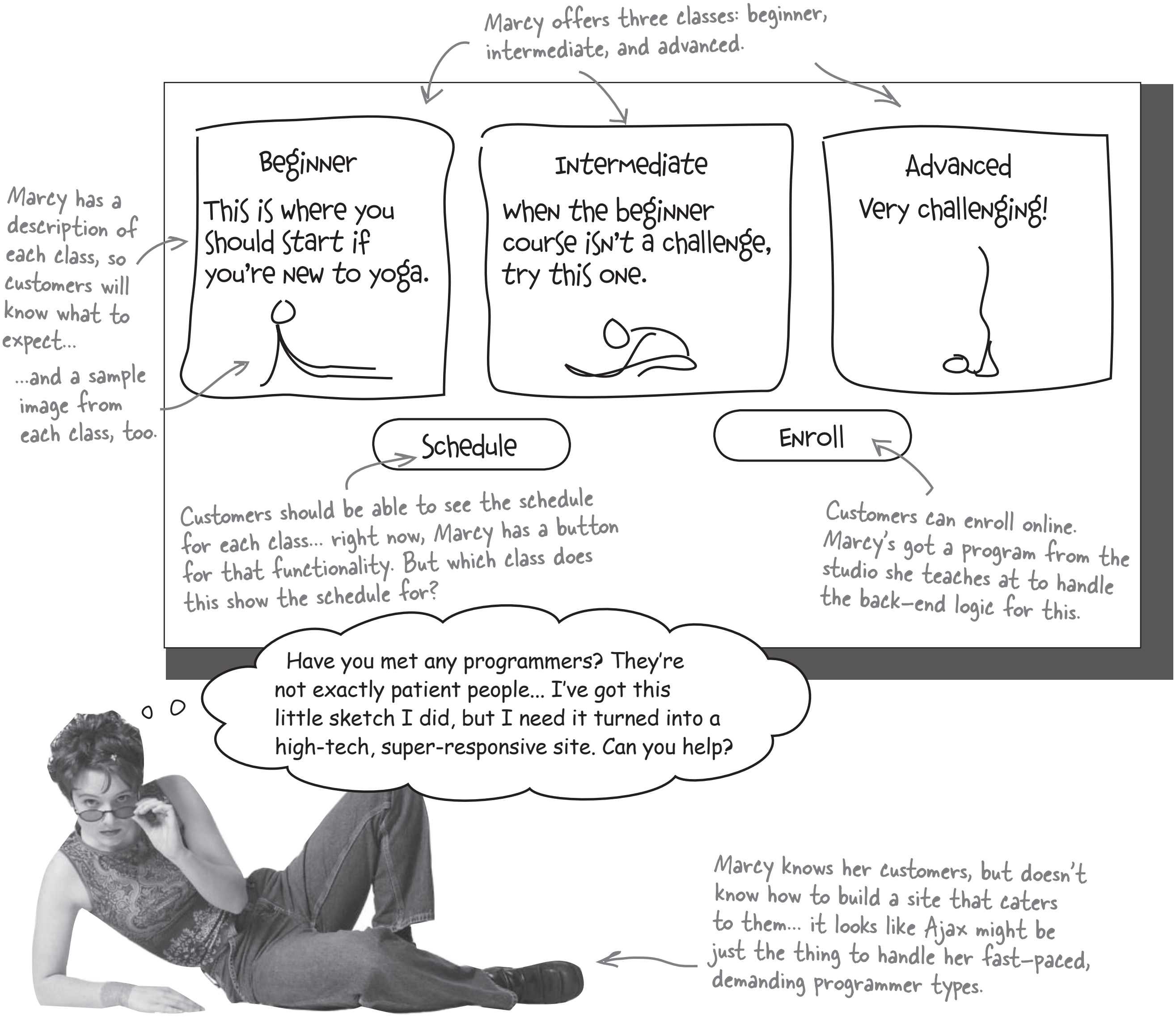


Sometimes you need your code to react to other things going on in your web application... and that's where **events** come into the picture. An event is *something that happens* on your page, in the browser, or even on a web server. But it's not enough to just know about events... sometimes you want to respond to them. By creating code, and registering it as an **event handler**, you can get the browser to run your handler every time a particular event occurs. Combine events and handlers, and you get **interactive web applications**.

It all started with a downward-facing dog...

Marcy's just opened up a new yoga studio that caters to programmers and techies. She wants a website that shows the different levels of classes she offers, the times they're available, and provides new customers a way to enroll in a class... all in a cool, intuitive package. But she doesn't have a clue how to build that kind of site.... that's where you come in.

To give you an idea of what she's looking for, Marcy worked up a quick sketch of what a page on her site needs to show her customers:

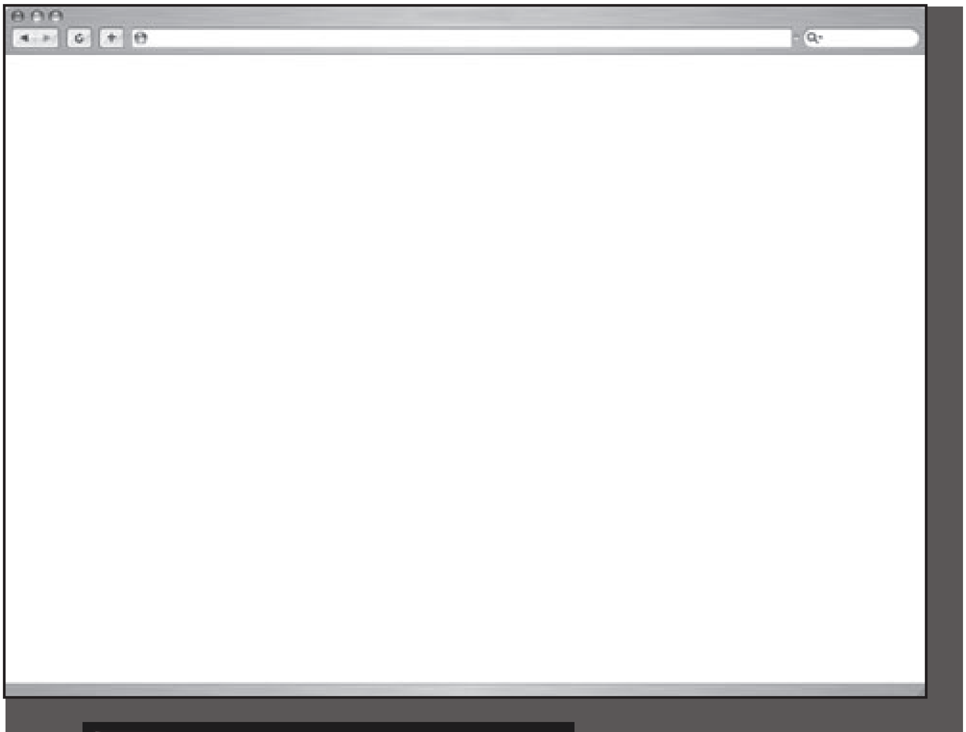


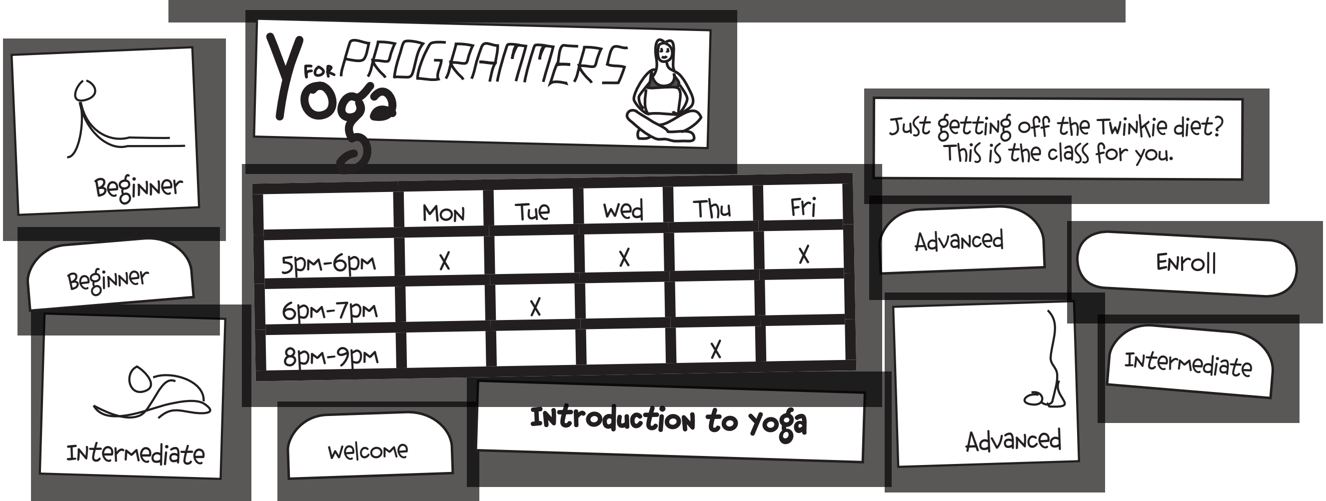


Design Magnets

It's time to turn Marcy's rough design into something you'd like to see in your own web browser (you're a programmer too, remember?). At the bottom of the page are lots of magnets representing parts of a potential web page. It's your job to arrange these on the web browser in a way that looks great. Also, see if you can design the site so when a customer clicks on a class level, the description and a picture of ***just the selected class*** is displayed.

In the rest of this chapter, you'll build this site, so think about what Ajax lets you do: change parts of the page in real-time, talk to the server asynchronously, etc.





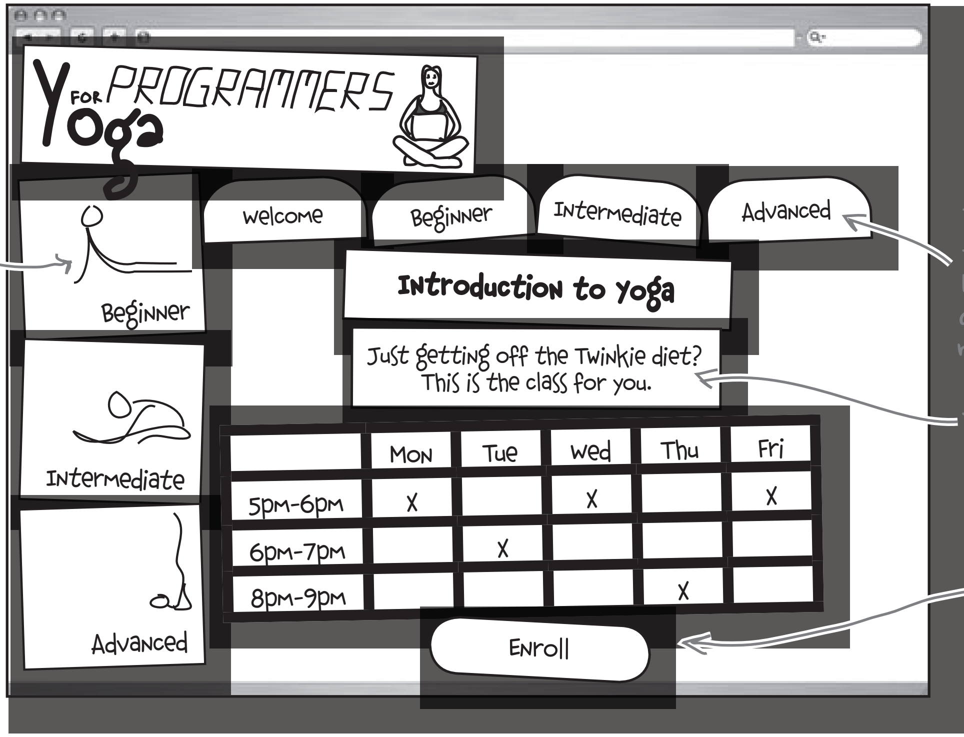


Our Design Magnet Solution

Your job was to arrange the design magnets onto the web browser in a way that looks great. Also, you should have designed the site so that when a customer clicks on a class level, the schedule, description, and a picture of ***just the selected class*** is displayed.

Here's where the Ajax came in... can we change out parts of the page without a reload? Of course...

These images will display a description of the class when the user rolls the mouse over them.



The "tabs" contain the class schedule. We'll get that on demand using a request object.

...and a description of the class will be displayed here.

When the schedule is displayed, clicking the Enroll button will take the user to a new page.

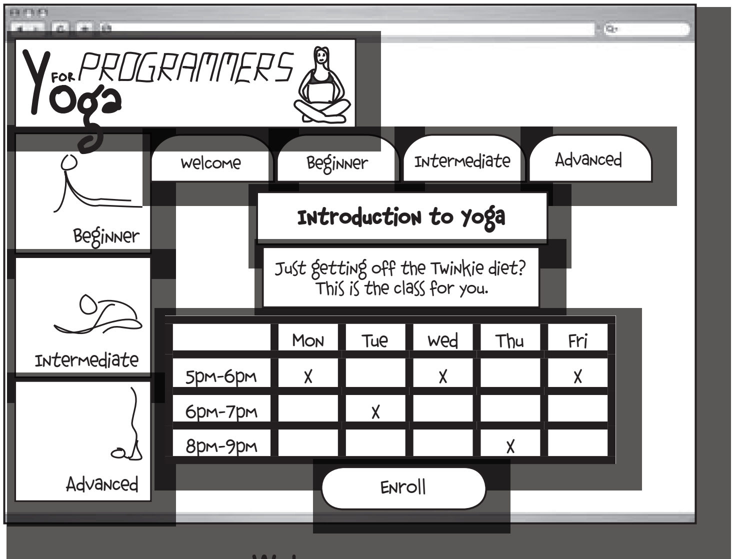
* It's okay if you came up with something a bit different, or even A LOT different... as long as you put together a dynamic design that doesn't require a reload to show each of the class schedules.

This chapter will be working off the design shown here, but feel free to change things up and implement your design instead...



Exercise

Suppose you want to build the website shown on page 96. In the space below, make notes about what should happen when customers click on different buttons, and what interactions you think you'll need between Marcy's web page and the server.



Web page



Web server

The studio Marcy teaches at has server-side programmers that can build whatever we need... but we've got to tell them what to build. What sort of server-side programs will we need?..

....

....

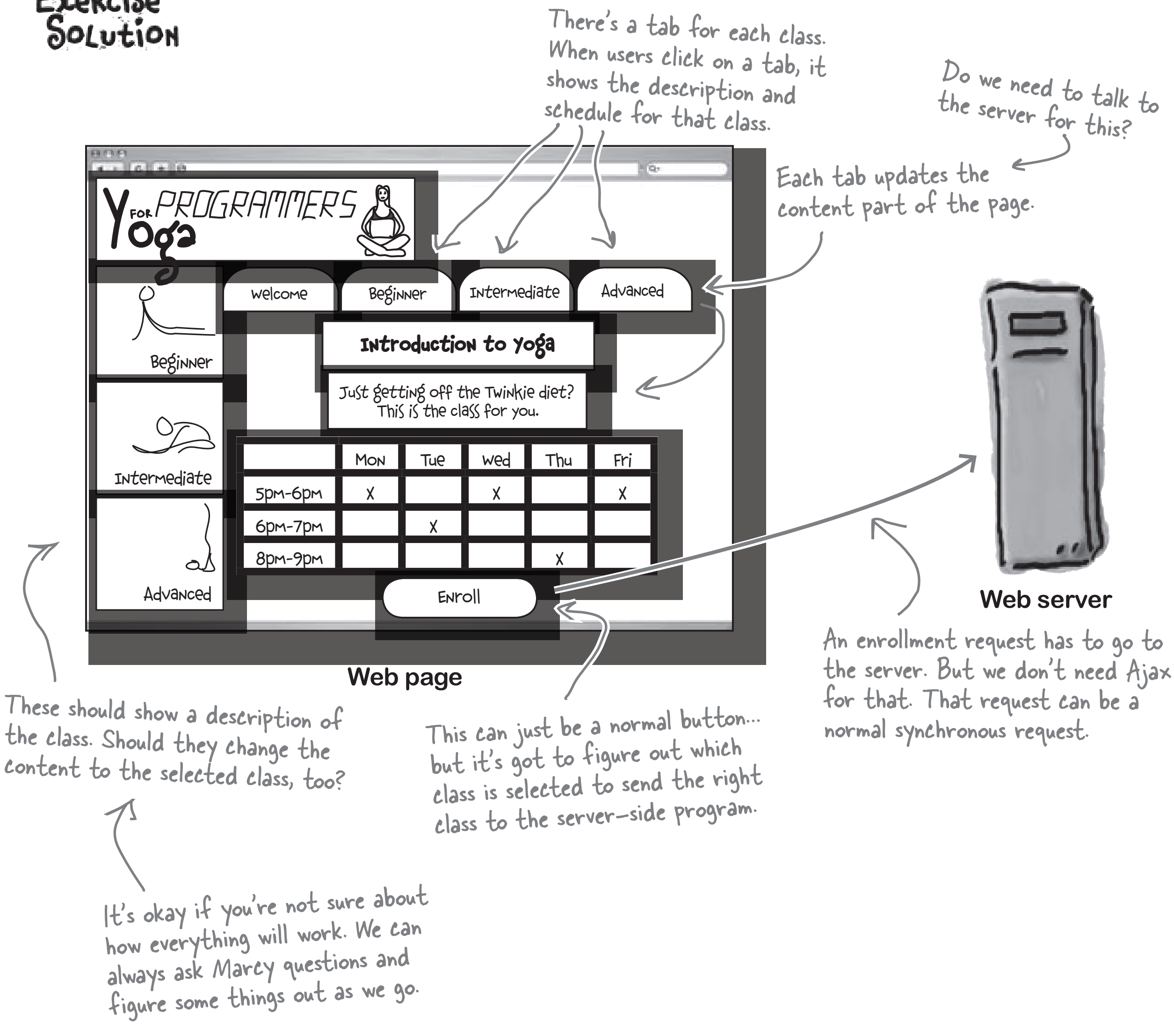
....

....

....



Your job was to make notes about what should happen when customers click on different buttons, and what interactions you think you'll need between Marcy's web page and the server.



The studio Marcy teaches at has server-side programmers that can build whatever we need... but we've got to tell them what to build. What sort of server-side programs will we need? *We need a program that takes a class name, and brings up some sort of enrollment form. It sounds like Marcy's got the form, so when users click a button, we send a request to the server for enrollment in the selected class.*

And what about the different class pages? We might need to request those from the server. Although something seems funny about that... I'm not sure what yet...

there are no Dumb Questions

Q: I came up with something totally different for my solution. Is that okay?

A: It is as long as you figured out that you need to make a request to the server to enroll for each different yoga class, and realized that each tab should bring up a class schedule and description. But some details are still fuzzy. What do those buttons on the left do? And do we need an asynchronous request to get information about each class?

Q: Are those tabs along the top of that drawing?

A: They sure are. Tabs are a great way to let users not only see different options, but easily click on each option to find out more.

Q: XHTML doesn't have a tab control. Do we have to buy a third-party tool or something?

A: No, we'll do it all using graphics, some nifty client-side JavaScript, and a request object to fetch the schedule from the server.

Q: But there are some toolkits that do that stuff for you, right? Why don't we just use one of those?

A: Toolkits are great, but it's much better to know what's going on underneath something like script.aculo.us or mootools. In this chapter, you'll build a simple tabbed control on your own... and then when you want to use a toolkit, you'll know what's going on, instead of depending on someone else to figure out your JavaScript for you.

And of course, when the toolkit you're using doesn't do just what you want, you'll be able to change it or write your own controls.

script.aculo.us and mootools are two popular JavaScript toolkits for visual effects, among other things.

Q: This doesn't look like much new... didn't we do something similar with the movie review site already?

A: That's right, you did. Although in that case, there was a lot less interactivity on the web page. Users signed up, and the page and your code did everything else. On this site, we've got a lot more going on: dealing with several different button presses, figuring out *which* button was pressed... loads of new interactivity issues. That's what most of this chapter focuses on, in fact: **events** and **interactivity**.

Try not to rely on any toolkit unless you understand the code behind that toolkit.

That way, when things don't work the way you want, you can fix the problems yourself.



Ajax IS mostly JavaScript, events, and lots of boring scripting!

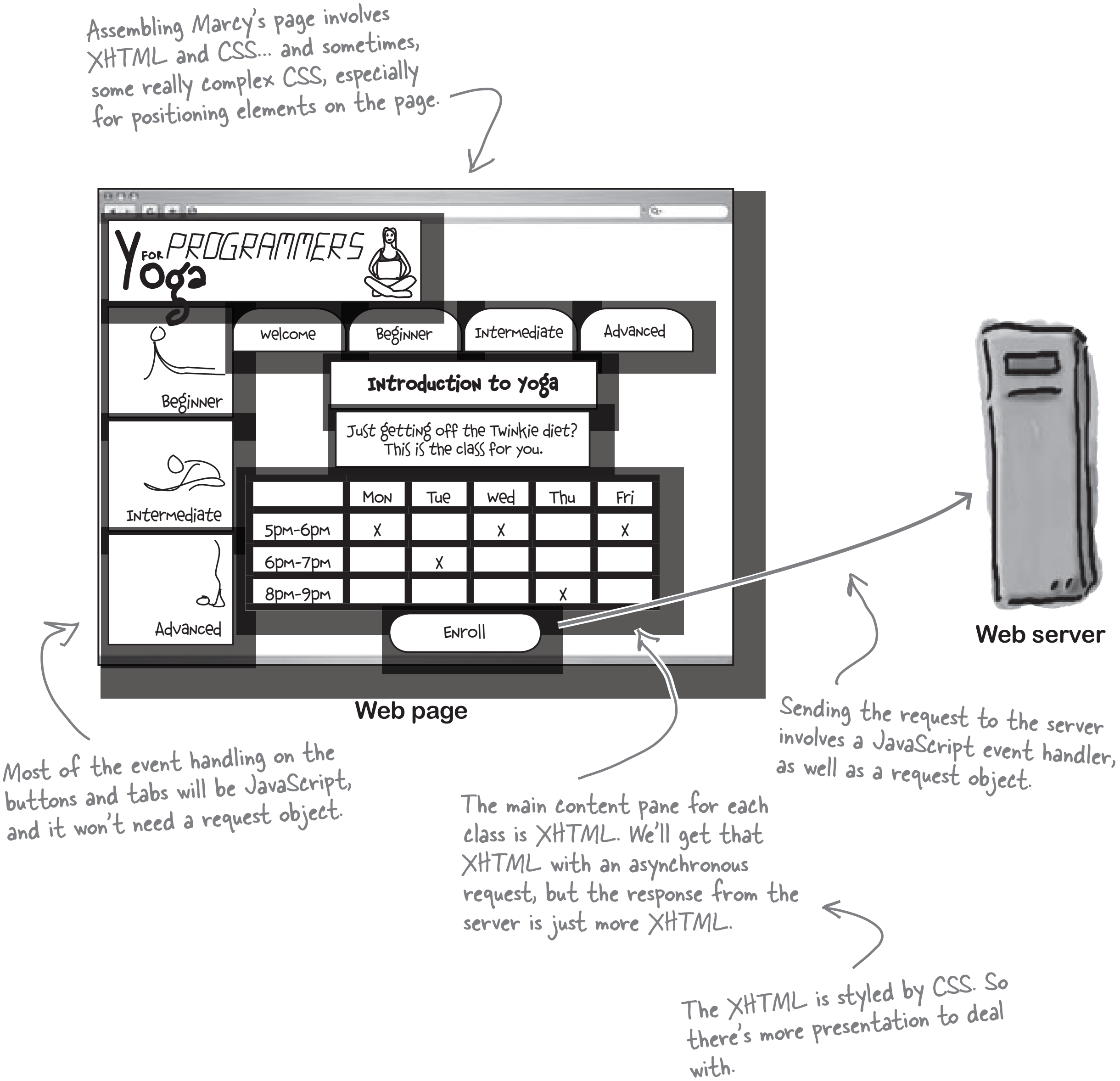
Most apps that use asynchronous requests have a lot ***more*** code that deals with web pages, objects on that page, and doing basic JavaScript tasks. The actual request object code may only be a callback function and a few lines in an event handler.

But you really can't break an application up into "JavaScript" and "Ajax" and "CSS." It all works together. So even though you'll be spending a lot of time in this chapter working with XHTML and CSS and event handlers, you're really building Ajaxian applications: responsive, user-friendly, modern web apps.

The more you know about JavaScript, XHTML, and CSS, the more effective and usable your Ajax apps will be.

Ajax apps are more than the sum of their parts

Ajax apps are really just a combination of lots of pretty simple technologies: XHTML, CSS, JavaScript, and things like the DOM, which you'll get to in a few chapters. In fact, if you take a close look at Marcy's app, ***most*** of the work is not Ajax-specific. It's XHTML, CSS, and JavaScript... with a little asynchronous requesting added in just when it's needed.



Here's Marcy's XHTML...

Below is the XHTML for Marcy's page... it's already got a few references to the JavaScript files we'll need and several <div>s representing the different parts of the page. Go ahead and download this page, along with the rest of the Chapter 3 examples, from the Head First Labs web site.

```
<html>
<head>
  <title>Yoga for Programmers</title>
  <link rel="stylesheet" href="css/yoga.css" type="text/css" />
  <script src="scripts/utils.js" type="text/javascript"></script>
  <script src="scripts/schedule.js" type="text/javascript"></script>
</head>

<body>
  <div id="schedulePane">
    <img id="logo" alt="Yoga for Programmers" src="images/logo.png" />
    <div id="navigation">
      <img src="images/beginnersBtn.png" alt="Beginners Yoga"
        title="beginners" class="nav" />
      <img src="images/intermediateBtn.png" alt="Intermediate Yoga"
        title="intermediate" class="nav"/>
      <img src="images/advancedBtn.png" alt="Advanced Yoga"
        title="advanced" class="nav"/>
     </div>

    <div id="tabs">
      <img src="images/welcomeTabActive.png" title="welcome" class="tab" />
      <img src="images/beginnersTabInactive.png" title="beginners" class="tab" />
      <img src="images/intermediateTabInactive.png"
           title="intermediate" class="tab"  />
      <img src="images/advancedTabInactive.png" title="advanced" class="tab" />
    </div>

    <div id="content">
      <h3>Click a tab to display the course schedule for the selected class</h3>
    </div>
  </div>
</body>
</html>
```

utils.js is the utility file we created in Chapter 2 with createRequest().

We'll be adding a few new functions to utils.js in this chapter.

schedule.js will store the application-specific JavaScript.

The working section of the page is wrapped in the "schedulePane" div.

This div contains the images on the left side of the page.

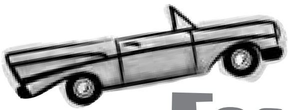
This div contains the four graphics that represent the "tabs."

Here's where we need to update the class information and display a schedule for each class.

classes.html, along with yoga.css and the images used by the Yoga web page, are all online at the Head First Labs web site.



classes.html



Test Drive

See what Marcy's page looks like pre-Ajax.

Download the examples for Chapter 3 from the Head First Labs web site. Go ahead and open up `classes.html`, and see what Marcy's page looks like. There's no interactivity yet, and you may get a message letting you know that `schedule.js` can't be found. That's okay, we'll get to all that soon.

These are just images that look like tabs. It's all just XHTML and CSS.



These really do look like buttons... we'll have to make sure they react like users expect them to.

Here's where a description of each class should appear.

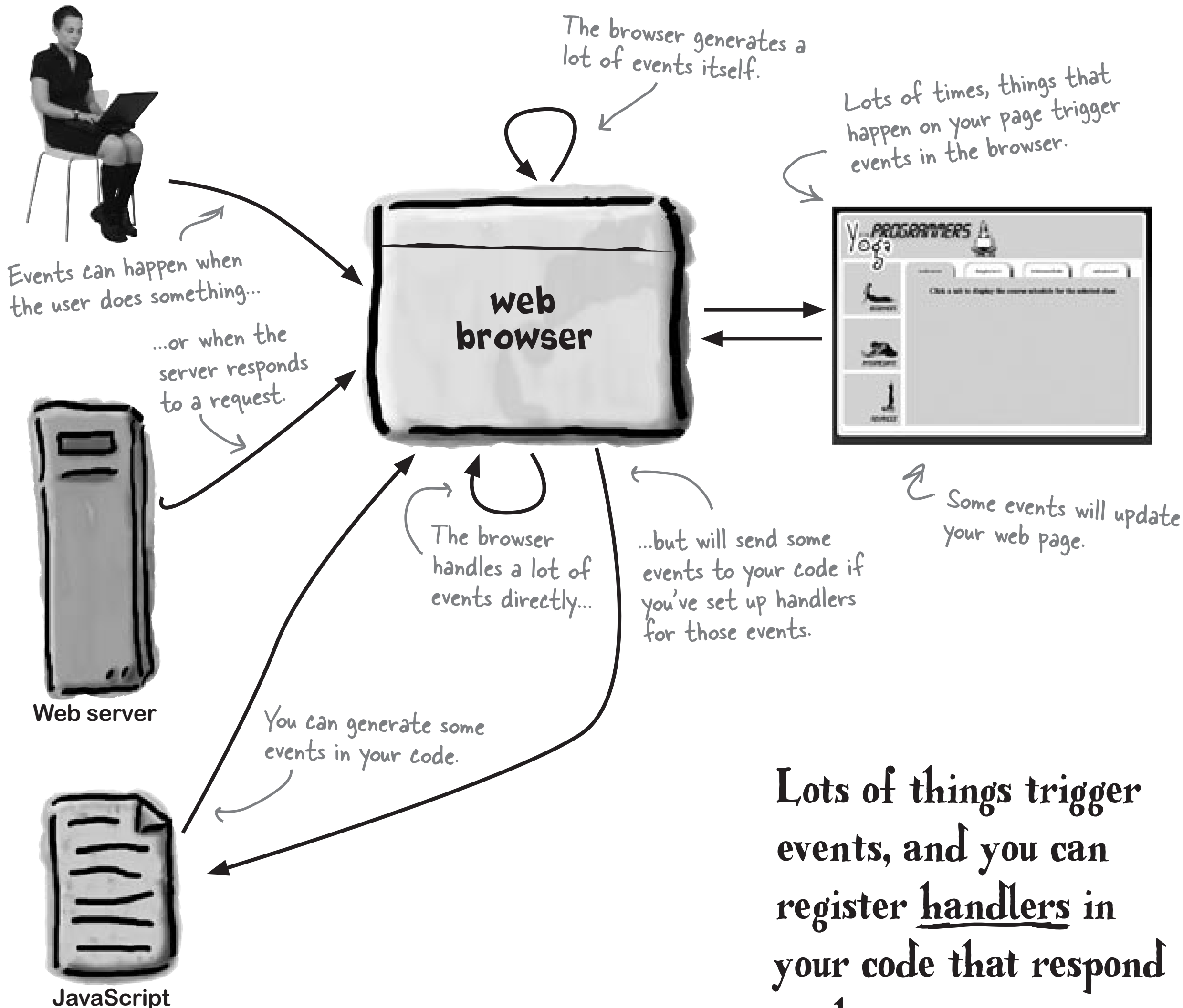
This looks a lot like the sketch on page 96. Now we just have to figure out the interactivity part.

Events are the key to interactivity

Marcy's page needs to react to her customers. She wants a different schedule and description to appear when a customer clicks on a class, and we could even highlight a menu item by using context-specific graphics.

Context-specific graphics is just a fancy term for changing a graphic when the customer moves their mouse over a menu option.

All this adds up to an **interactive web page**. In programming terms, "interactive" means that your page responds to specific **events**. And events are just things that happen. Those things can be triggered by the user, your code, the browser, or even a server:



Lots of things trigger events, and you can register handlers in your code that respond to those events.

Who Does What?

The names of most events give you a pretty good idea what they do.
Can you match the events to what they might be used for?

onclick	Use me when you want to validate a form before its contents are processed by a server-side program.
onfocus	Use me when you want to provide audio feedback when a user has images disabled.
onblur	Use me when you want to scroll in on a portion of an image when a user clicks a certain spot of the image.
onload	Use me when you want to let users know that increasing the width of the browser window could reduce their viewing experience.
onmouseover	Use me when you want to hide a submenu when the user moves away from a menu item.
onmouseout	Use me when you want to let a user know about the input format for a selected text field.
onsubmit	Use me when you want to change the color of a menu item when a user hovers over the item.
onresize	Use me when you want to give the user popup instructions for a form before they start using the form.
onerror	Use me when you want to validate a particular field every time data is entered into that field.

Who Does What?

Solution

The names of most events give you a pretty good idea what they do. Your job was to match the events to what they might be used for.

onclick

onfocus

onblur

onload

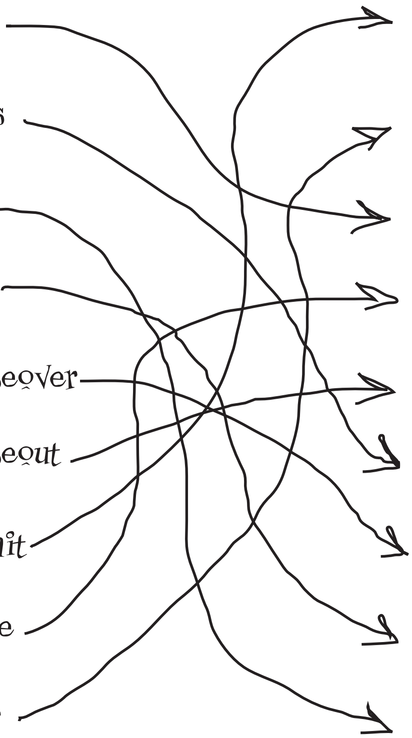
onmouseover

onmouseout

onsubmit

onresize

onerror



You want to validate a form before its contents are processed by a server-side program.

You want to provide audio feedback when an image can't be loaded.

You want to scroll in on a portion of an image when a user clicks a certain spot of the image.

You want to let users know that increasing the width of the browser window could reduce their viewing experience.

You want to hide a submenu when the user moves away from a menu item.

You want to let a user know about the input format for a selected text field.

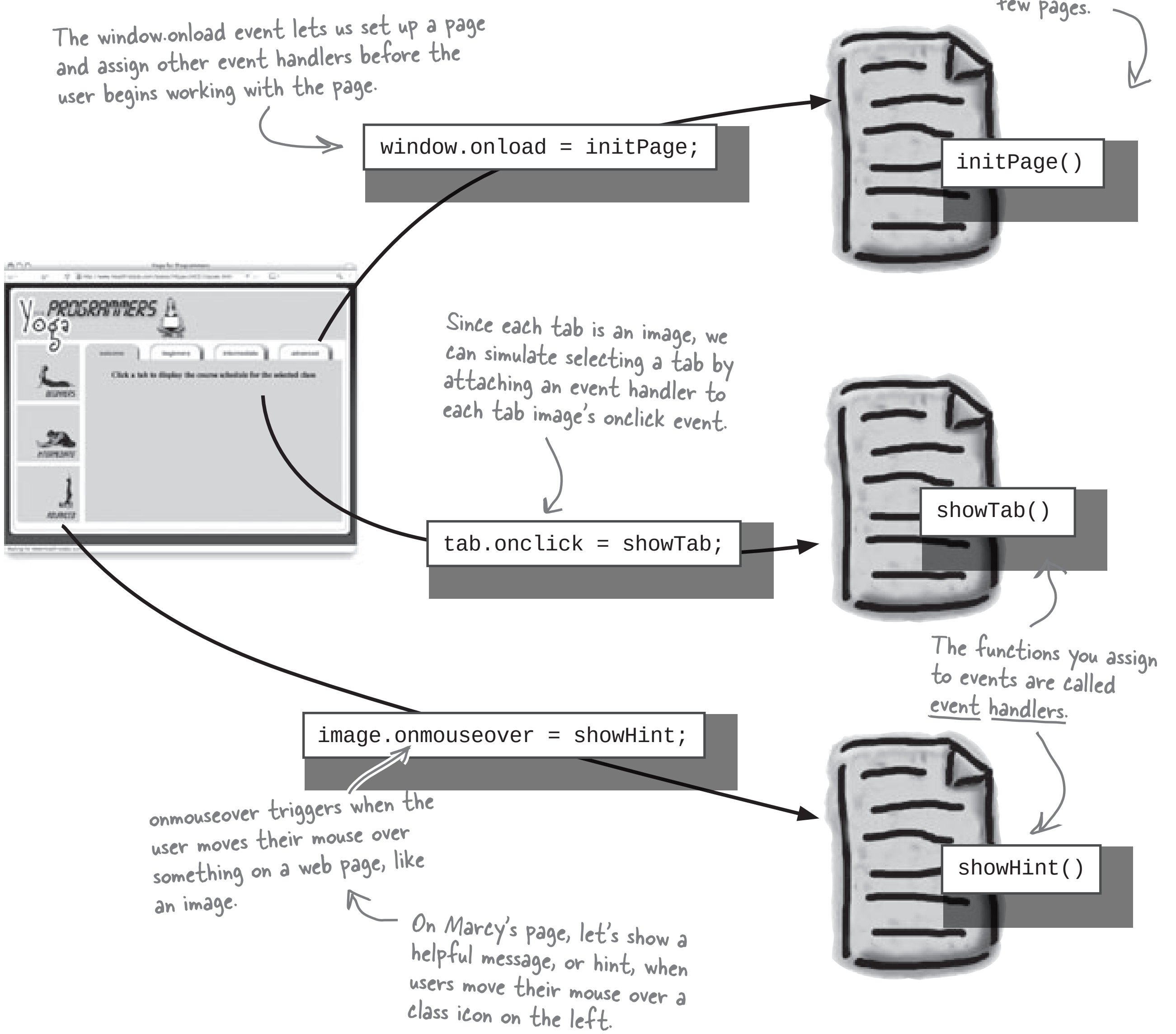
You want to change the color of a menu item when a user hovers over the item.

You want to give the user popup instructions for a form before they start using the form.

You want to validate a particular field every time data is entered into that field.

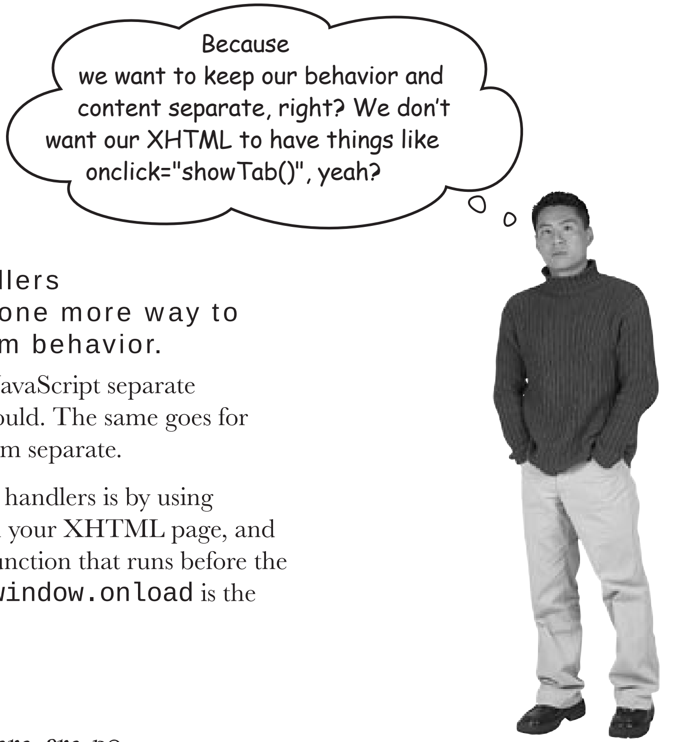
Connect events on your web page to event handlers in your JavaScript

You've already used the `window.onload` event to trigger lots of setup work on web pages, and the `onclick` event to handle users clicking on an image. We can use these events, as well as the `onmouseover` event, to connect different parts of Marcy's yoga page to JavaScript functions we'll write.



Use the window.onload event to initialize the rest of the interactivity on a web page

You've already used `window.onload` twice to initialize a page. We need to do the same thing in Marcy's yoga page because...



Assigning event handlers programmatically is one more way to separate content from behavior.

Anytime you can keep your JavaScript separate from your XHTML, you should. The same goes for XHTML and CSS: keep them separate.

The best way to assign event handlers is by using properties of the elements in your XHTML page, and doing that assignment in a function that runs before the user gets control of a page. `window.onload` is the perfect event for just that.

there are no Dumb Questions

Q: So when does window.onload get called again?

A: Actually window.onload is an event. The event occurs, or fires, once the XHTML page has been read by the browser and all the files that XHTML references have been loaded.

Q: So when window.onload fires, the browser runs that event's handler function?

A: Exactly.

Q: How does the browser know what function to call?

A: The browser will call the function that you assign to the `onload` property of the `window` object. You set that property just like any other JavaScript property: with an equals sign. Just be sure you leave any parentheses off the name of the function:

```
window.onload = initPage;
```

Q: And we assign that property where?

A: The browser will run any code that isn't inside a function as soon as it encounters that code. So just put the window.onload assignment at the top of your JavaScript, outside of any function, and the assignment will happen before the user can interact with your page.

Then, the browser will fire onload and run the function you just assigned. That's your chance to set up the web page's other events.



JavaScript Magnets

You need to initialize Marcy's yoga page. Each left-side image and tab should display information about the rolled-over class. Additionally, clicking on a tab should select the class that's clicked. See if you can use the magnets below to build initPage(), as well as placeholders for the other functions referenced. For now, the placeholders can just display alert boxes.

HINT: One magnet goes here, before initPage().

```
function initPage() {
```

* For now, Marcy just wants the tabbed images clickable. The images on the left side should show a hint when the user rolls a mouse over them, but they shouldn't do anything else.

```
initPage
for (var i=0; i<images.length; i++)
function initPage()
function showHint()
window.onload
function hideHint()
"
"
"
currentImage.onmouseover =
var currentImage = images[i];
currentImage.onclick =
)
)
)
;
;
;
alert(
alert(
}
}
}
}
}
}
}
{
{
{
{
{
{
currentImage.onmouseout =
=
in showHint()
in hideHint()
var images = document.getElementById("schedulePane").getElementsByTagName("img");
showTab
alert(
function showTab()
if (currentImage.className = "tab")
in showTab()
hideHint
"
"
"
showHint
;
;
;
;
```



JavaScript Magnet Solution

Using the steps on page 16 and what you've learned about how events work in JavaScript, can you re-create the initialization code for Marcy's page?

```
window.onload = initPage;

function initPage() {
  var images = document.getElementById("schedulePane").getElementsByTagName("img");
  for (var i=0; i<images.length; i++) {
    var currentImage = images[i];
    currentImage.onmouseover = showHint;
    currentImage.onmouseout = hideHint;
    if (currentImage.className = "tab") {
      currentImage.onclick = showTab;
    }
  }
}

function showHint() {
  alert(" in showHint()");
}

function hideHint() {
  alert(" in hideHint()");
}

function showTab() {
  alert(" in showTab()");
}
```

This line sets the initPage() function to be called once the XHTML page has been completely loaded.

Don't worry too much about this one. We'll talk about all these getElement methods in Chapters 5 and 6.

Loop over every single <img> element, and attach events to each.

Did you remember this opening brace?

Get a reference to the current <img>.

These events will happen when the user moves the mouse over one of the class thumbnails or a tab.

When a mouse rolls over an image, show a hint...

...and hide the hint when the user mouses off of the image.

Remember not to include the parentheses. You want to reference the function, not run it.

Did you remember to close everything off?

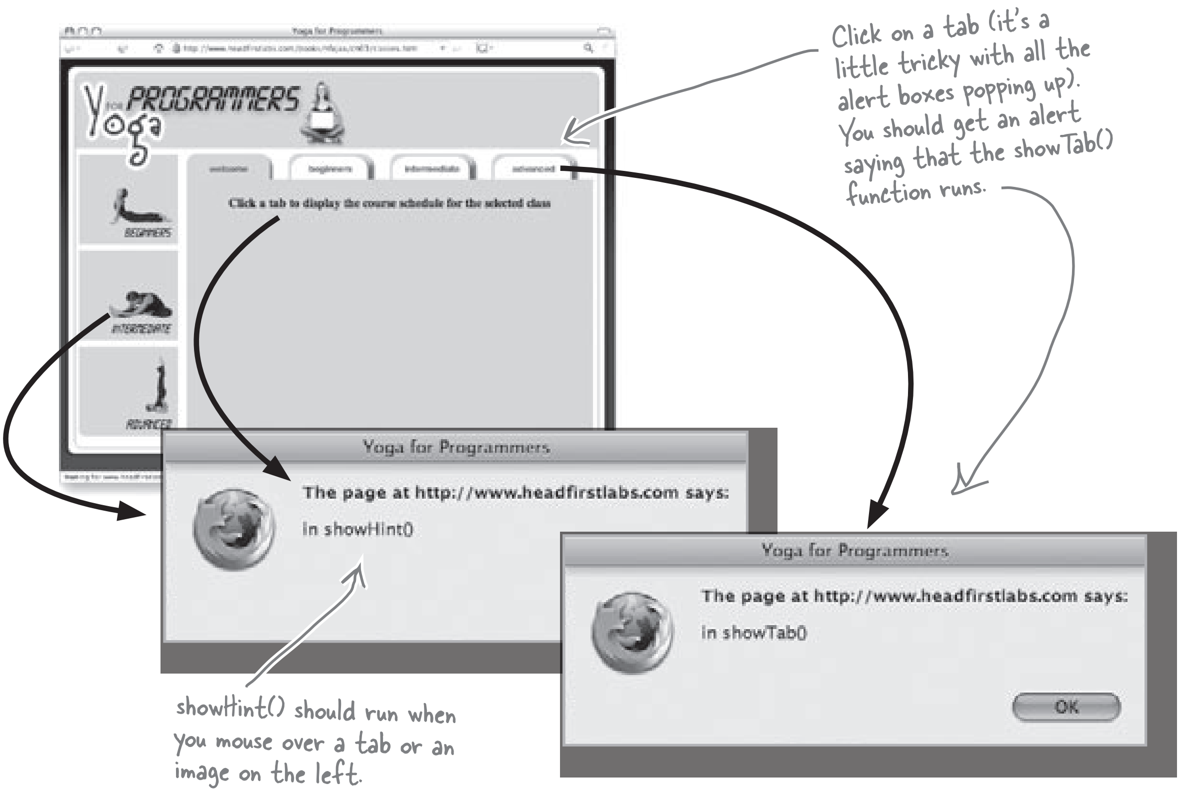
Right now Marcy only wants the tab images to be clickable.

These functions can be declared in any order. You just need the declarations to avoid a JavaScript error.

Test Drive

Create schedule.js, and add the functions shown on the last page. Don't forget to assign initPage to the window.onload event, too. Then, test things out in your web browser.

Roll your mouse over a tab. You should see an alert for `showHint()`, and then `hideHint()`. Try and click on a tab, too. Do you get a `showTab()` alert? What about clicking on an image on the left? Nothing should happen there right now.



Brain Power

How do you like this user interface? Is there anything you would change?

Look, you can say what you want, but I still think those images over on the left look like buttons. Navigation's almost always on the left. Are we just going to ignore like a hundred years of web design?



If a web page is confusing to YOU, it will almost certainly be confusing to your users.

When you design and implement a site, you know how the site is supposed to act. If the site's confusing to you, then users—without the benefit of the information you have—will probably be even more confused.

Even when you don't get to control the design of a site, like with Marcy's yoga page, you should still try and make the site as unconfusing as possible. If that means turning some images into buttons to avoid users clicking on those images endlessly, then do it!

But isn't that confusing, too? If the images are clickable, then you've got two forms of navigation: tabs and images.



Sometimes you have to make a choice between okay and better.

This can even happen when you design a site that a customer loves. Later on, you realize there are some problems, but the customer doesn't want to make any changes because they like what they've already seen. →

If you're not in control of a site's design, you're often stuck making the best decisions based on an existing layout. With Marcy's site, she liked the design with tabs and images.

Your job, then, is to make the best decisions based on what you've got. In this case, that means two forms of navigation to avoid user-confusion. Otherwise, non-clickable images on the left might be construed as buttons.

Change those left-side images to be clickable

It's a piece of cake to make the images on the left-hand side of Marcy's page clickable. In fact, all we need to do is *remove* code:

```
function initPage() {
  var images =
    document.getElementById("schedulePane").getElementsByTagName("img");
  for (var i=0; i<images.length; i++) {
    var currentImage = images[i];
    currentImage.onmouseover = showHint;
    currentImage.onmouseout = hideHint;
    if (currentImage.className=="tab") {
      currentImage.onclick = showTab;
    }
  }
}
```

We don't want just the tabs to be clickable... we want all images, including the left-hand ones, to be clickable.

Don't forget to remove the closing brace.

window.onload = initPage; urlHeader=....

schedule.js



Try it out... each image should call showTab() now.



Good web pages aren't confusing

A good web page is as intuitive as possible. If something looks like a button, make it a button. And if part of a site is confusing to you, the web programmer, it's probably even more confusing to a user.

Use your XHTML's content and structure

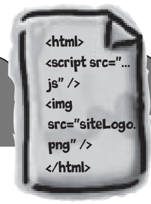
`showHint()` is called when a user rolls their mouse over a tab or image. But how do we know which tab or image is being rolled over? For that, we need to take another look at Marcy's XHTML:

```
... XHTML for page head and body...
  <div id="schedulePane">
    <img id="logo" alt="logo" src="images/logo.png" />
    <div id="navigation">
      <img src="images/beginnersBtn.png" alt="Beginners Yoga"
        title="beginners" class="nav" />
      <img src="images/intermediateBtn.png" alt="Intermediate Yoga"
        title="intermediate" class="nav"/>
      <img src="images/advancedBtn.png" alt="Advanced Yoga"
        title="advanced" class="nav"/>
    </div>

    <div id="tabs">
      <img src="images/welcomeTabActive.png" title="welcome" class="tab" />
      <img src="images/beginnersTabInactive.png" title="beginners" class="tab" />
      <img src="images/intermediateTabInactive.png"
          title="intermediate" class="tab" />
      <img src="images/advancedTabInactive.png" title="advanced" class="tab" />
    </div>
```

Each image has a title attribute that identifies the class level.

The tab graphics use the same titles.

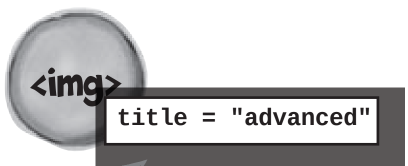


classes.html

Every XHTML element is accessible in your JavaScript code as an object

You've been using `getElementById()` to access the images in Marcy's XHTML page. That works because each element in the XHTML is represented by the browser as an object you can manipulate in your JavaScript.

Even better, all the attributes on an element are stored as properties on the JavaScript object that represents that element. Since Marcy's images have titles, we can use those titles to figure out which image or tab was selected and show the right hint.



The title attribute becomes a property of the JavaScript object that represents the <img> element.

Sharpen your pencil

showHint() should display a short hint-style message about each class when a user rolls their mouse over an image. But the hint should only appear if the welcome tab is selected; if one of the classes is selected, hints are disabled. Your job is to complete the code for showHint() that's started below.

```
var welcomePaneShowing = ................;
```

This is a global variable: it's outside any functions. It should indicate if the welcome pane is showing, which is the only time we want to show hints.

```
function showHint() {
  alert("in showHint()");
  if (!............................. ) {
    return;
  }

  switch (this.....................) {
    case "................................":
      var hintText = "Just getting started? Come join us!";
      break;
    case "................................":
      var .............. = "Take your flexibility to the next level!";
      break;
    case " ..............................":
      var hintText =
        "Perfectly join your body and mind with these intensive workouts.";
       ...................
    .............. :
      var .............. = "Click a tab to display the course schedule for the class";
  }
  var contentPane = ....................................("content");
  ................ .innerHTML = "<h3>" + .............. + "</h3>";
}
```

(The line `alert("in showHint()");` is struck through on the page.)

Sharpen your pencil Solution

Your job was to complete the showHint function to display a hint based on the title of the image.

```
var welcomePaneShowing = true;

function showHint() {
  alert("in showHint()");
  if (!welcomePaneShowing ) {
    return;
  }
  switch (this.title) {
    case "beginners":
      var hintText = "Just getting started? Come join us!";
      break;
    case "intermediate":
      var hintText = "Take your flexibility to the next level!";
      break;
    case "advanced":
      var hintText =
        "Perfectly join your body and mind with these intensive workouts.";
      break;
    default :
      var hintText = "Click a tab to display the course schedule for the class";
  }
  var contentPane = getElementById("content");
  contentPane.innerHTML = "<h3>" + hintText + "</h3>";
}
```

(The line `alert("in showHint()");` is struck through.)

When the page loads, the welcome pane is showing. So we start out with this set to true.

Make sure this variable is declared outside of initPage(), showHint(), or any other function.

If we're not on the welcome page, don't do anything. Just return.

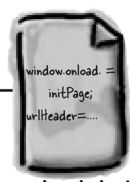
"this" refers to whatever object called this function. That's the image that the user rolled over.

"title" is the attribute of the XHTML page we want to check... so we access it with the "title" property of the image.

For each different class level, we want to set some hint text.

It's always good practice to have a default in your switch statements. Our default can just be a generic instruction message.

All that's left is to get the <div> where the content is shown, and show the hint text.



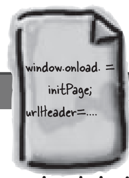
schedule.js

Add the code for hideHint(), too

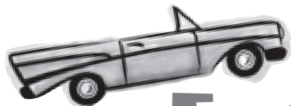
The code for `hideHint()` is simple once `showHint()` is done. You just need to grab the content pane, and set the hint text back to the default:

```
function hideHint() {
  alert("in hideHint()");
  if (welcomePaneShowing) {
    var contentPane = document.getElementById("content");
    contentPane.innerHTML =
      "<h3>Click a tab to display the course schedule for the class</h3>";
  }
}
```

This is basically the reverse of showHint(). The function grabs the content <div> and sets the text to the default message.

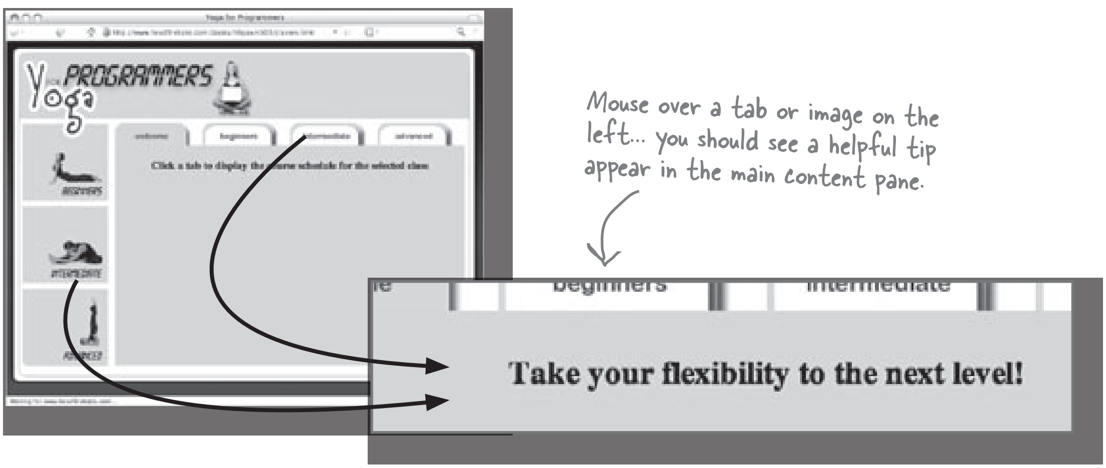


schedule.js



Test Drive

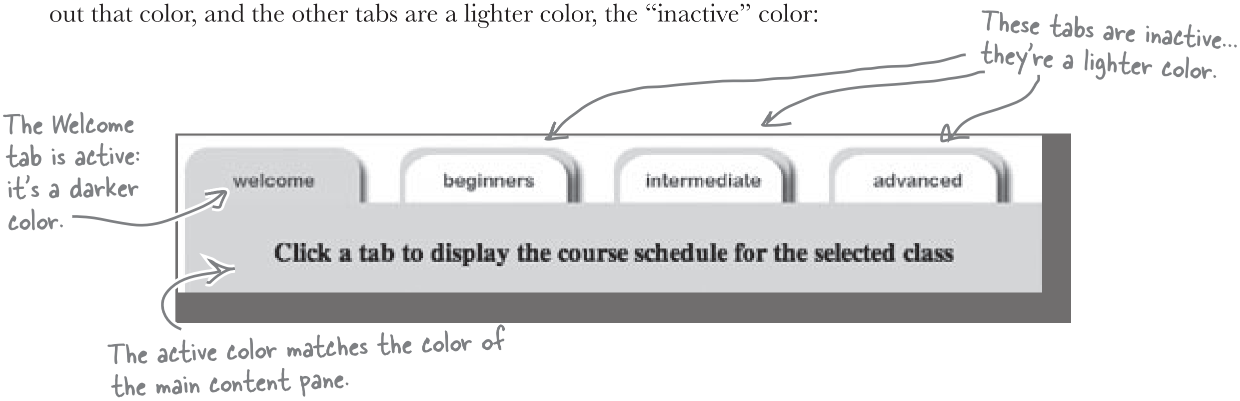
Update schedule.js. Add a welcomePaneShowing variable, and update the showHint() and hideHint() functions. Then try everything out.



Tabs: an optical (and graphical) illusion

Marcy likes the look and feel of tabs on her yoga page. While there are lots of fancy toolkits that let you create tabs, a simple graphical trick is all we need.

On the yoga page, we've got a main content pane that's dark green. So that color basically becomes the "active" color. The Welcome tab starts out that color, and the other tabs are a lighter color, the "inactive" color:



To make a tab active, we need to change the tab's background to the "active" color

All we need to do to make a different tab active is change it to the active color. Then, we can make the old active tab inactive by changing it to the inactive color.

So suppose we've got two graphics for each tab: one with the tab against an active background, and another with the tab against an inactive background:



We've already got a `showTab()` function. So the first thing that function should do is change the tab image for the clicked-on tab.

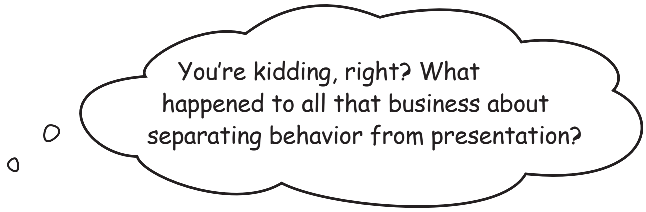


Use a for... loop to cycle through the images

You've already used the `title` property of the image objects in `showHint()` to change the hint text. We need to do something similar in `showTab()`: figure out which tab should be active, and change that tab to the active image. For all the other tabs, we just want the inactive image.

```
function showTab() {
  alert("in showTab()");
  var selectedTab = this.title;

  var images = document.getElementById("tabs").getElementsByTagName("img");
  for (var i=0; i<images.length; i++) {
    var currentImage = images[i];
    if (currentImage.title == selectedTab) {
      currentImage.src = "images/" + currentImage.title + "Top.png";
    } else {
      currentImage.src = "images/" + currentImage.title + "Down.png";
    }
  }
}
```





This event handler has a <u>LOT</u> of presentation-specific details.

`showTab()` now works directly with image names, and it actually builds those image names dynamically! So not only does `showTab()` mix behavior with presentation (the images), but it actually is depending on the content of the XHTML page—the title of each image—to figure out what presentation to use.

There's a real problem here. But what would ***you*** do to fix the problem? Think about how you would separate content, presentation, and behavior ***before you turn the page.***

CSS classes are the key (again)

Marcy likes the look and feel of tabs on her yoga page. While there are lots of fancy toolkits that let you create tabs, a simple graphical trick is all we need.

For each tab, there are two possible states: active, which is the darker color that matches the content pane, and inactive, which is the lighter, unselected color. So we can build two CSS classes for each tab: one active, and one inactive.

Open up `yoga.css` in your app's `css/` directory, and add these lines:

```
#tabs a#welcome.active {
  background: url('../images/welcomeTabActive.png') no-repeat;
}
#tabs a#welcome.inactive {
  background: url('../images/welcomeTabInactive.png') no-repeat;
}
#tabs a#beginners.active {
  background: url('../images/beginnersTabActive.png') no-repeat;
}
#tabs a#beginners.inactive {
  background: url('../images/beginnersTabInactive.png') no-repeat;
}
#tabs a#intermediate.active {
  background: url('../images/intermediateTabActive.png') no-repeat;
}
#tabs a#intermediate.inactive {
  background: url('../images/intermediateTabInactive.png') no-repeat;
}
#tabs a#advanced.active {
  background: url('../images/advancedTabActive.png') no-repeat;
}
#tabs a#advanced.inactive {
  background: url('../images/advancedTabInactive.png') no-repeat;
}
```

Each tab has an active class with the active image...

...and an inactive class with the inactive tab image.

There are two classes for each tab.

This CSS can go anywhere in yoga.css. It's up to you.



yoga.css

Ummm... but the tabs aren't <a>'s!

Did you notice what element the CSS is styling? `#tab` indicates a `<div>` with an id of "tab." That's okay. But then, the CSS indicates it's styling `<a>` tags, with ids of "welcome," "beginners," and so on. That doesn't match Marcy's XHTML page at all.

But that's no big deal... we can change all the images on the XHTML page to `<a>` tags to separate one more layer of content from presentation.

```
... XHTML for page head, body, and schedulePane <div>...

    <div id="tabs">
      <a id="welcome" title="welcome" class="active" href="#">Welcome</a>
      <a id="beginners" title="beginners" class="inactive" href="#">Beginners</a>
      <a id="intermediate" title="intermediate" class="inactive"
         href="#">Intermediate</a>
      <a id="advanced" title="advanced" class="inactive" href="#">Advanced</a>

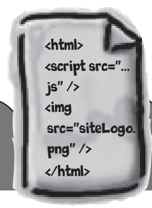
      <img src="images/welcomeTabActive.png" title="welcome" class="tab" />
      <img src="images/beginnersTabInactive.png" title="beginners" class="tab" />
      <img src="images/intermediateTabInactive.png"
           title="intermediate" class="tab" />
      <img src="images/advancedTabInactive.png" title="advanced" class="tab" />
    </div>
```

The elements still have a title and an id.

Now we can indicate which tabs are active, and which are inactive.

Each tab is now represented by an <a> element.

Remove the direct references to images in the XHTML.



classes.html

there are no Dumb Questions

Q: Why is the href set to "#"?

A: # references the current page. We don't want the tabs to take the user anywhere else, although later we'll write code so that clicking on a tab shows the selected class's schedule.

Q: If we're not taking the user anywhere, why use <a> elements?

A: Because the tabs are ultimately links. They link to each class schedule, even if it's in a slightly non-traditional way. So the best XHTML element for a link is `<a>`.

On the other hand, there are usually at least two or three ways to do something on the Web. You could use a `<span>` element, a `<div>`, or even an image map. It's really up to you. As long as you can attach event handlers to the element, you're good to go.

This broke our JavaScript, too, didn't it?

We've got a nice, clean XHTML page and some CSS that truly controls the presentation of the page. But now all that JavaScript that depended on `<img>` elements isn't going to work. That's okay, though, because even though it worked before, it mixed images in with behavior... a real problem.

Let's fix up our script, and separate all that presentation from the behavior of the yoga page.

```
window.onload = initPage;
var welcomePaneShowing = true;

function initPage() {
  var tabs =
    document.getElementById("tabs").getElementsByTagName("a");
  for (var i=0; i<tabs.length; i++) {
    var currentTab = tabs[i];
    currentTab.onmouseover = showHint;
    currentTab.onmouseout = hideHint;
    currentTab.onclick = showTab;
  }

  var images =
    document.getElementById("schedulePane").getElementsByTagName("img");
  for (var i=0; i<images.length; i++) {
    var currentImage = images[i];
    currentImage.onmouseover = showHint;
    currentImage.onmouseout = hideHint;
    currentImage.onclick = showTab;
  }
}

function showHint() {
  // showHint() stays the same
}
function hideHint() {
  // hideHint() stays the same
}
```

We grab the tabs <div>, and iterate over the <a> elements.

This isn't much different... the events and handlers are the same as when the tabs were images.

We need a new block here because iterating over images won't get the tabs... they're now <a> elements.

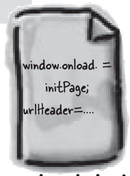
This code looks awfully similar to the code up above, and we already know that repeated code can be a real trouble spot. We might need to come back to this later.

```
function showTab() {
  var selectedTab = this.title;

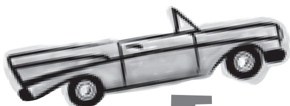
  var tabs = document.getElementById("tabs").getElementsByTagName("a");
  for (var i=0; i<tabs.length; i++) {
    var currentTab = tabs[i];
    if (currentTab.title == selectedTab) {
      currentTab.className = 'active';
    } else {
      currentTab.className = 'inactive';
    }
  }
}
```

This is the same loop as in initPage(). We want all the <a> tags in the tabs <div>.

No image names. Now we just change out the CSS classes. Much better!

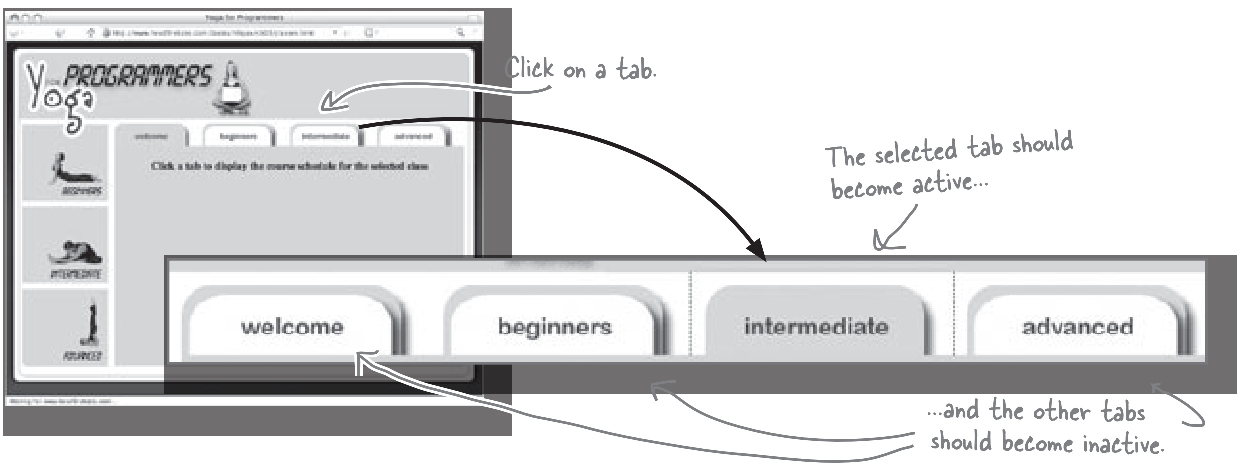


schedule.js



Test Drive

You've got a lot of changes to make. Update classes.html, yoga.css, and schedule.js. Then, see if those tabs work... try clicking on each.





The earlier you separate your content from your presentation and behavior, the easier that separation will be.

With Marcy's site, we didn't really think about content or presentation early on, and it's only when we were a few functions into our behavior (the JavaScript) that we saw problems. If we'd planned from the start to keep images out of our code, and let our CSS handle all the presentation, we'd have fewer changes to make to our JavaScript.

Still, even if you find problems late in the process, it's almost always better to do the work to really separate out content, presentation, and behavior. Your app will be a lot better for the work you put into it.

there are no Dumb Questions

Q: So should I never have images in my XHTML? That's basically what we did with the tabs, right? Pulled the <img> elements out of the XHTML?

A: That was part of it. But more importantly, we used CSS to control whether or not a button was active. What a button looks like when it changes from active to inactive is presentation, so that belongs in the CSS.

It's okay to have images in your XHTML; just make sure that if those images are tied to behavior, you get your CSS and code involved, and keep those details out of your XHTML.

Q: That CSS confused me. What does "#tabs a#advanced.inactive" mean?

A: The # sign indicates an id. So `#tabs` means "anything with an id of 'tabs'." In the XHTML, that's the `<div>` with an id of "tabs."

Then, `a#advanced` means "for an <a> element with an id of 'advanced'." So that's the `<a>` element with an id of "advanced" nested within a `<div>` with an id of "tabs."

And finally, the "`.`" indicates a class. So `a#advanced.inactive` means the `<a>` element with an id of "tabs" and a class name of "advanced" (all under a `<div>` with an id of "tabs"). That's a mouthful, so if you're still unsure about the CSS, you might want to pick up a copy of Head First HTML with CSS & XHTML to help you out.

Q: Isn't it sort of weird that all the buttons on the left are images, but all the tabs are <a> elements? Why aren't we using <a> elements for the buttons, too?

A: Good question. We'll come back to that, but anytime you notice things that seem out of place, jot down a note to yourself. It might be something worth looking at in more detail.

Q: When I click a button on the left, the tab also changes. Is that right?

A: What do you think? When you select the "advanced" button, do you think the "advanced" tab should become active?

Sharpen your pencil

We've gotten a lot done, but showTab() is still incomplete. We've got to show the schedule for a selected class when a tab is clicked on. Assume the schedule is an HTML description and a table that shows the days of the week that the selected class is available. There will also probably be an "Enroll button."

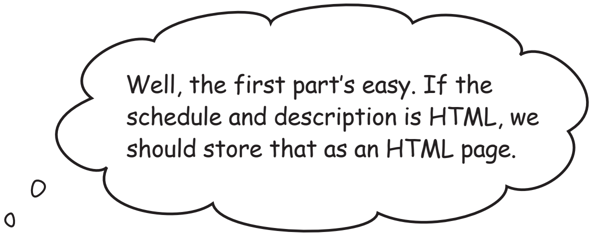
How should we store the details and schedule for each class? In an HTML file? In the JavaScript? Why did you choose the format you did? ....

How would you replace the main content pane with the schedule and details for a selected class? ....

Does your solution:

- [ ] Separate the content of your page from its presentation?
- [ ] Separate the content of your page from its behavior?
- [ ] Separate the behavior of your page from its presentation?

Ajax is all about <u>INTERACTION</u>. Your page can interact with server-side programs, elements within itself, and even other pages.





Jill: You mean ***X***-HTML, right?

Joe: Well, yeah.

Frank: As an entire page? That's no good... we don't want to recreate all the tabs and stuff for each class, do we?

Joe: Well, then how about an XHTML fragment. Like, just the elements and text for the actual schedule and class description.

Jill: Yeah, because there's no way we want all that content in our JavaScript. And if we use XHTML, we can use the same CSS styles as in the main page.

Frank: But how do we load up that... what? XHTML fragment—

Joe: Sure.

Frank: —okay, right. So how do we load it?

Joe: Well, the tabs are `<a>` elements. Maybe we put the fragments in the `href` attributes instead of those # symbols?

Frank: But that would replace the entire page. That's no good. Besides, it seems sort of slow...

Jill: Guys, what about using a request object?

Joe: What do you mean?

Jill: What if we use a request object to get the XHTML fragment, and just set the content pane's `innerHTML` to the returned page?

Frank: Can you even do that?

Jill: Why not? Instead of requesting a server-side program, we'll just request the XHTML fragment we want.

Joe: And we can do it asynchronously, so there's no waiting or page refreshing!

Use a request object to fetch the class details from the server

The server doesn't need to do any processing for Marcy's page, but we can still use a request object to grab the XHTML fragments for each class. This is a request to the server, but it's just for a page rather than a program. Still, the details are the same as you've already seen.

We'll build the code the same way we always do, using the `createRequest()` function from `utils.js` and a callback to display the results in the content pane. Here's what we need:



schedule.js

```
function showTab() {
  var selectedTab = this.title;

  // set each tab's CSS class

  var request = createRequest();
  if (request==null) {
    alert("Unable to create request");
    return;
  }
  request.onreadystatechange = showSchedule;
  request.open("GET", selectedTab + ".html", true);
  request.send(null);
}

function showSchedule() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      document.getElementById("content").innerHTML =
        request.responseText;
    }
  }
}
```

This is the part of the showTab() function that you've already written.

This is the same request creation code we've been using for talking to server-side programs

This time we're sending the request object to a page URL. So we need to name the fragments beginner.html, intermediate.html, and advanced.html.

The showSchedule() callback function is called when the request returns.

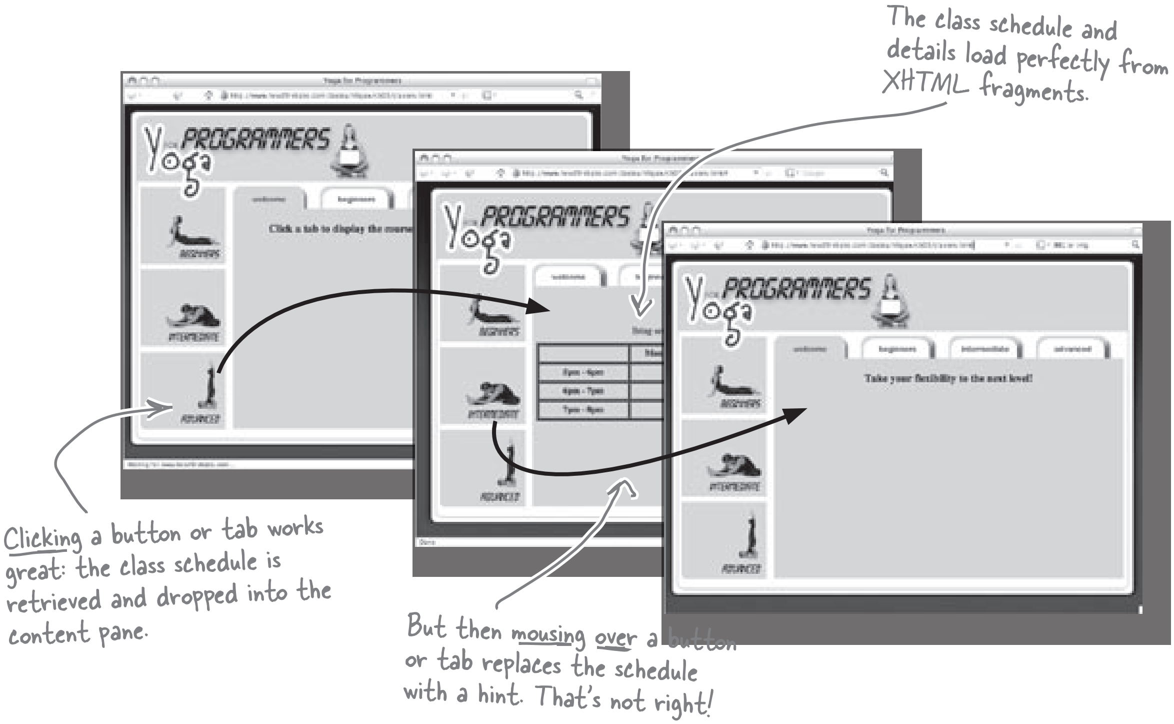
The XHTML in the file is available in responseText. We can display the XHTML using the innerHTML property of the content page.

BRAIN POWER

Make these changes to schedule.js, and try out the improved web page. Is everything working? Are there any changes you'd make?

Be careful when you have two functions changing the same part of a web page

There's a bug! Class schedules show up okay, but mousing over another tab or button image hides the class schedule and replaces the content pane with hint text. That doesn't seem too intuitive...





Make sure you have the class XHTML fragments.

XHTML fragments for each class are included in the examples download from Head First Labs. Make sure they're named `beginner.html`, `intermediate.html`, and `advanced.html`. They should be in the same directory as your main page, `classes.html`.

Sharpen your pencil

It's time to finish up Marcy's page. You need to change the JavaScript so that hints only show when the welcome tab is active. If a class is selected, no hints should appear. Mark up, cross out, and add to the code below to finish up schedule.js. To help you out, we've only shown the parts of the script that you need to change or add to, and parts that are relevant to those changes. Good luck!

```
var welcomePaneShowing = true;

function showHint() {
  if (!welcomePaneShowing) {
    return;
  }
  // code to show hints based on which tab is selected
}

function showTab() {
  var selectedTab = this.title;

  var tabs = document.getElementById("tabs").getElementsByTagName("a");
  for (var i=0; i<tabs.length; i++) {
    var currentTab = tabs[i];
    if (currentTab.title == selectedTab) {
      currentTab.className = 'active';
    } else {
      currentTab.className = 'inactive';
    }
  }
  var request = createRequest();
  if (request == null) {
    alert("Unable to create request");
    return;
  }
  request.onreadystatechange = showSchedule;
  request.open("GET", selectedTab + ".html", true);
  request.send(null);
}
```

Sharpen your pencil Solution

Your job was to finish up the code so that tabs selected classes and turned off hints. Hints should only appear when the Welcome tab is active.

```
var welcomePaneShowing = true;

function showHint() {
  if (!welcomePaneShowing) {
    return;
  }
  // code to show hints based on which tab is selected
}

function showTab() {
  var selectedTab = this.title;
  if (selectedTab == "welcome") {
    welcomePaneShowing = true;
    document.getElementById("content").innerHTML =
      "<h3>Click a tab to display the course schedule for the class</h3>";
  } else {
    welcomePaneShowing = false;
  }

  // everything else stayed the same!
}
```

This is the key variable. This should indicate if the welcome pane is showing. If it's not, we don't want to show any hints.

We've already got a check in showHint() for the welcome pane... so we just need to make sure this variable is set correctly.

Here's the new code. First, we need to see if the selected tab is the Welcome tab.

If so, we should update the welcomePaneShowing variable.

You get bonus credit if you figured this out. If the welcome pane is selected, we need to overwrite any class schedule with the welcome message...

...otherwise, the welcome pane will have a class schedule, and it won't even be clear which class is showing!

This really is a hack... we're putting presentation in our JavaScript! You'll learn a way to avoid this when you get into the DOM in Chapters 5 and 6.

If any other tab is selected, update welcomePaneShowing to false. Now no hints will show because of the if statement in showHint().

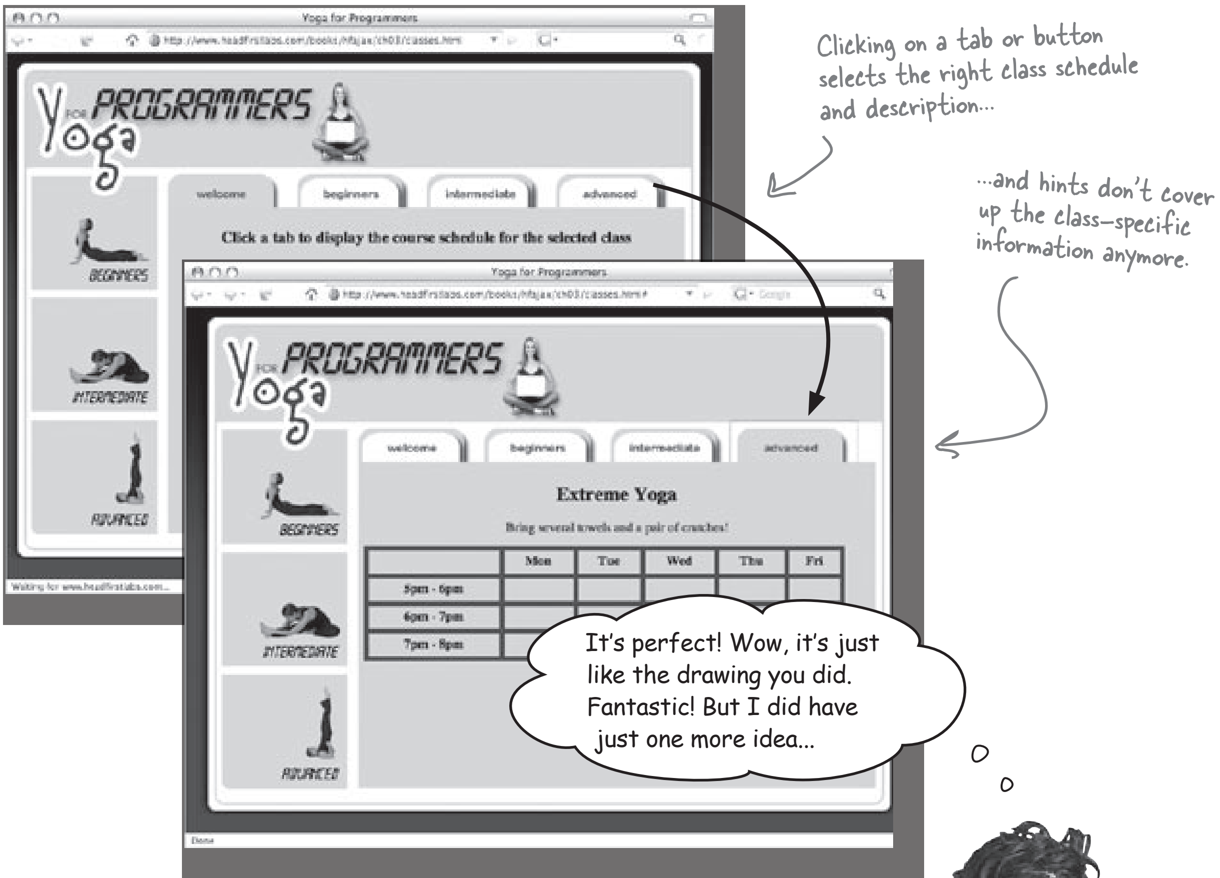
there are no Dumb Questions

Q: What if I didn't catch the bit about changing the content pane back to the welcome message when the Welcome tab is selected?

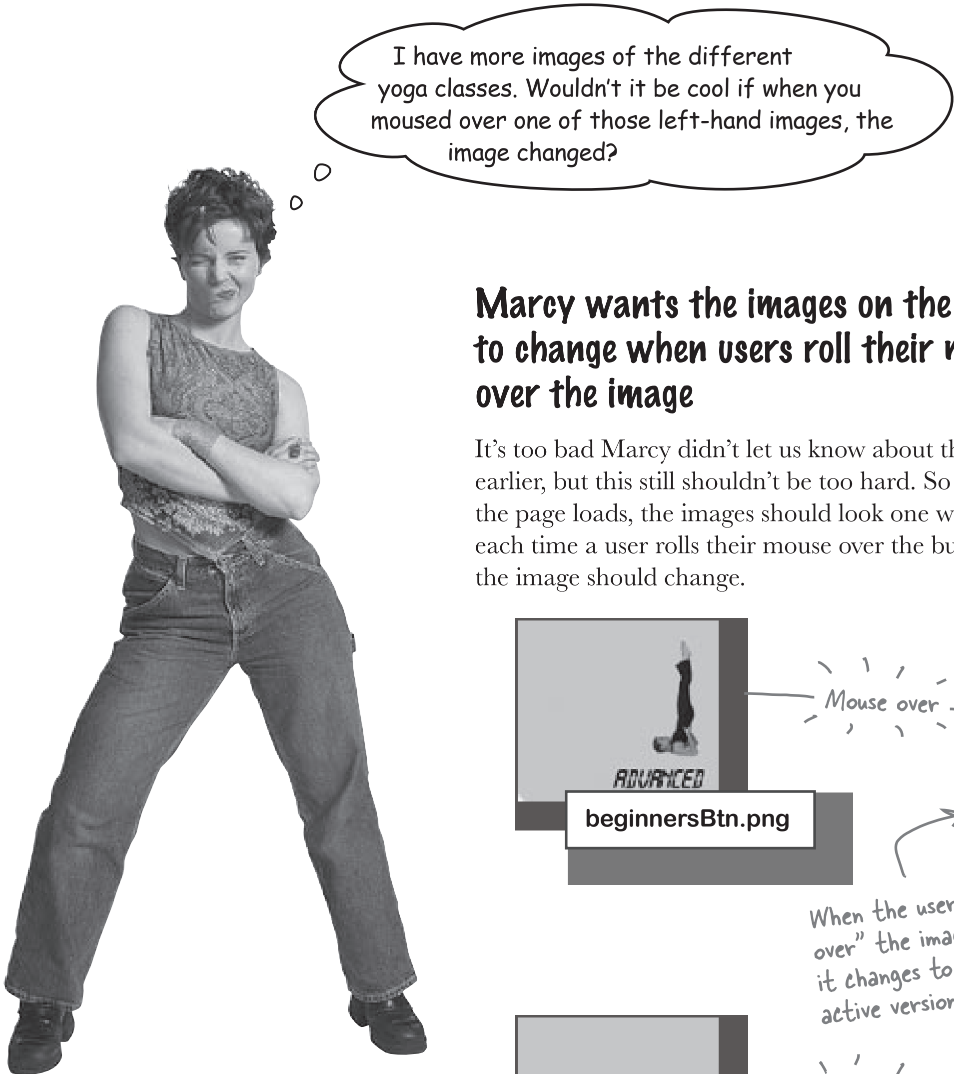
A: That's okay. Make sure you add that code to your copy of schedule.js, though. One way you can avoid missing things like that in the future is to always test your code. Load up the yoga class page, and click and move the mouse around... does anything look funny? If so, then make whatever changes you need to fix that problem.

Test Drive

Make sure you've got the XHTML fragments, the updated CSS, the clean XHTML class page (with no presentation!), and your completed copy of schedule.js. Load up Marcy's web page, and give it a spin.

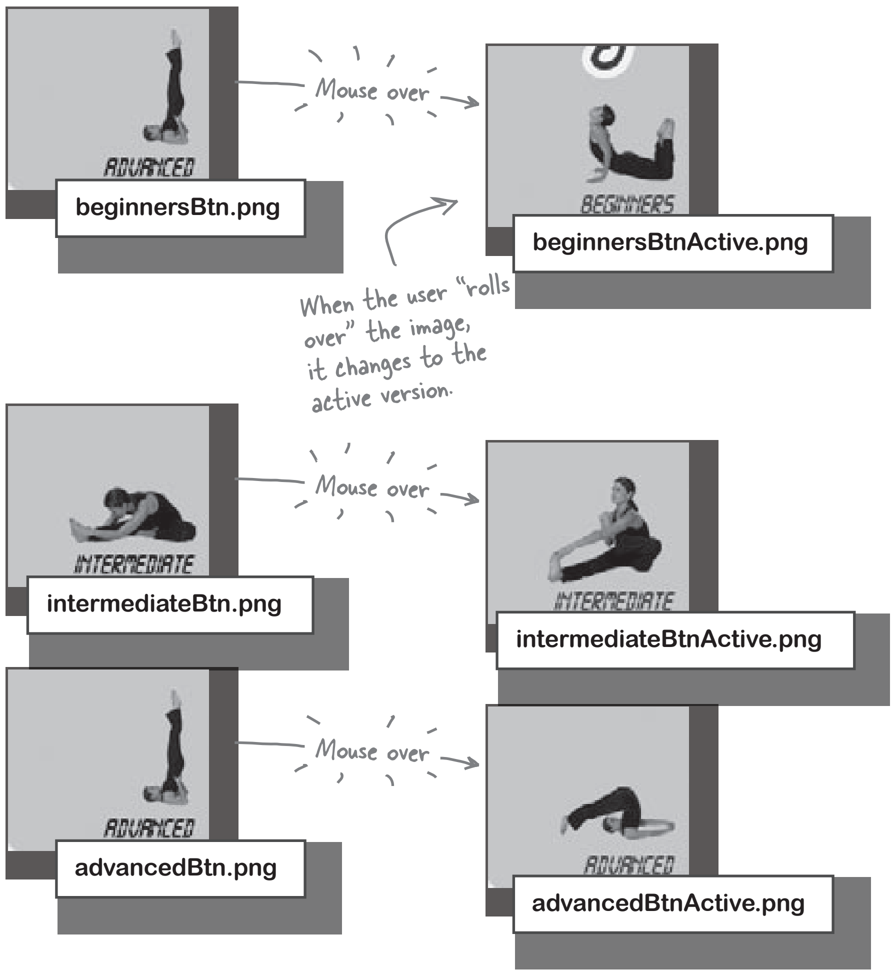






Marcy wants the images on the left to change when users roll their mouse over the image

It's too bad Marcy didn't let us know about this earlier, but this still shouldn't be too hard. So when the page loads, the images should look one way, but each time a user rolls their mouse over the button, the image should change.



When you need to change images in your script, think "change CSS classes" instead

Here's another case where separation of presentation and behavior can be a big issue. Before changing any code, think, "Is this a case where I'm going to have to mix my behavior (my code) and my presentation (like images)?"

If it is, it's time to restructure some things. The image buttons are really just like the tabs. They just look like buttons instead of tabs. So let's add some new CSS classes for both button states: the normal button and the active button.

```
#navigation a {
  display: block;  float: left;
  height: 0;  margin: 0 0 10px 0;
  overflow: hidden;  padding: 140px 0 0 0;
  width: 155px;  z-index: 200;
}

#navigation a#beginners {
  background: url('../images/beginnersBtn.png') no-repeat;
}
#navigation a#beginners.active {
  background: url('../images/beginnersBtnActive.png') no-repeat;
}
#navigation a#intermediate {
  background: url('../images/intermediateBtn.png') no-repeat;
}
#navigation a#intermediate.active {
  background: url('../images/intermediateBtnActive.png') no-repeat;
}
#navigation a#advanced {
  background: url('../images/advancedBtn.png') no-repeat;
}
#navigation a#advanced.active {
  background: url('../images/advancedBtnActive.png') no-repeat;
}
```

These rules apply to all the <a>'s and handle positioning and sizing.

By default, buttons use one image...

...and when a button is active, it uses a different image.



yoga.css

Just like with the classes for the tabs, these can go anywhere in your CSS.

Links in XHTML are represented by <a> elements

Here's another place where we can make some improvements to our XHTML. Currently, the images are represented by `<img>` tags, but they really are functioning as linking buttons: you can click on one to get a class schedule.

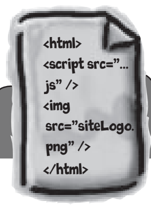
Let's change each button to an `<a>`, which better represents something you can click on to get to a different destination, in this case a class schedule and description.

```
... XHTML for page head, body, etc...

  <div id="navigation">
    <a id="beginners" title="beginners" href="#">Beginners</a>
    <a id="intermediate" title="intermediate" href="#">Intermediate</a>
    <a id="advanced" title="advanced" href="#">Advanced</a>
    <img src="images/beginnersBtn.png" alt="Beginners Yoga"
         title="beginners" class="nav" />
    <img src="images/intermediateBtn.png" alt="Intermediate Yoga"
         title="intermediate" class="nav"/>
    <img src="images/advancedBtn.png" alt="Advanced Yoga"
         title="advanced" class="nav"/>
  </div>
```

Make sure to use the right ids and titles.

This text will get covered up by the button images.



classes.html

there are no Dumb Questions

Q: With the tabs, we had an inactive class and an active class. But on the buttons, they're in the XHTML without a class, and then there's a CSS "active" class description with the active image. Why don't we have an inactive CSS class with these buttons, too?

A: Good question. With the tabs, there were two distinct states: active (in the forefront) and inactive (in the background). The buttons we have, though, really have a normal state, where they sit flat, and an active state, where the button is highlighted. So it seemed more accurate to have a button (with no class), and then assign that button the "active" class when it's rolled over. Uniformity is a good thing, though, so you could probably use inactive and active classes if you felt strongly about it.

We need a function to show an active button and hide a button, too

Before we change any of schedule.js, let's add two functions we know we'll need. First, we need a `buttonOver()` function to show the active image for a button. That's just a matter of changing a CSS class:

```
function buttonOver() {
  this.className = "active";
}
```

When the mouse is over a button, make it active.

We can do just the opposite for when a user's mouse rolls out of the button's area. We just need to change back to the default state, which is no CSS class:

```
function buttonOut() {
  this.className = "";
}
```

When the mouse rolls out of a button, go back to the default state.

In JavaScript, an element is represented by an object. That object has a property for each event that can occur on the element that it represents.

When you initialize the page, you need to assign the new event handlers

Now we need to assign the new functions to the right events. `buttonOver()` should get assigned to a button's onmouseover event, and `buttonOut()` gets assigned to a button's onmouseout event.

We can also update the code to use the new `<a>` elements that represent buttons instead of the older `<img>` elements.

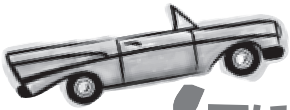
In our updated XHTML, we need all the <a> elements nested in the navigation <div>.

```
function initPage() {
  // code to deal with tabs

  var buttons =
    document.getElementById("navigation").getElementsByTagName("a");
  for (var i=0; i<buttons.length; i++) {
    var currentBtn = buttons[i];
    currentBtn.onmouseover = showHint;
    currentBtn.onmouseout = hideHint;
    currentBtn.onclick = showTab;
    currentBtn.onmouseover = buttonOver;
    currentBtn.onmouseout = buttonOut;
  }
}
```

We've changed the array that was called images to be called buttons.

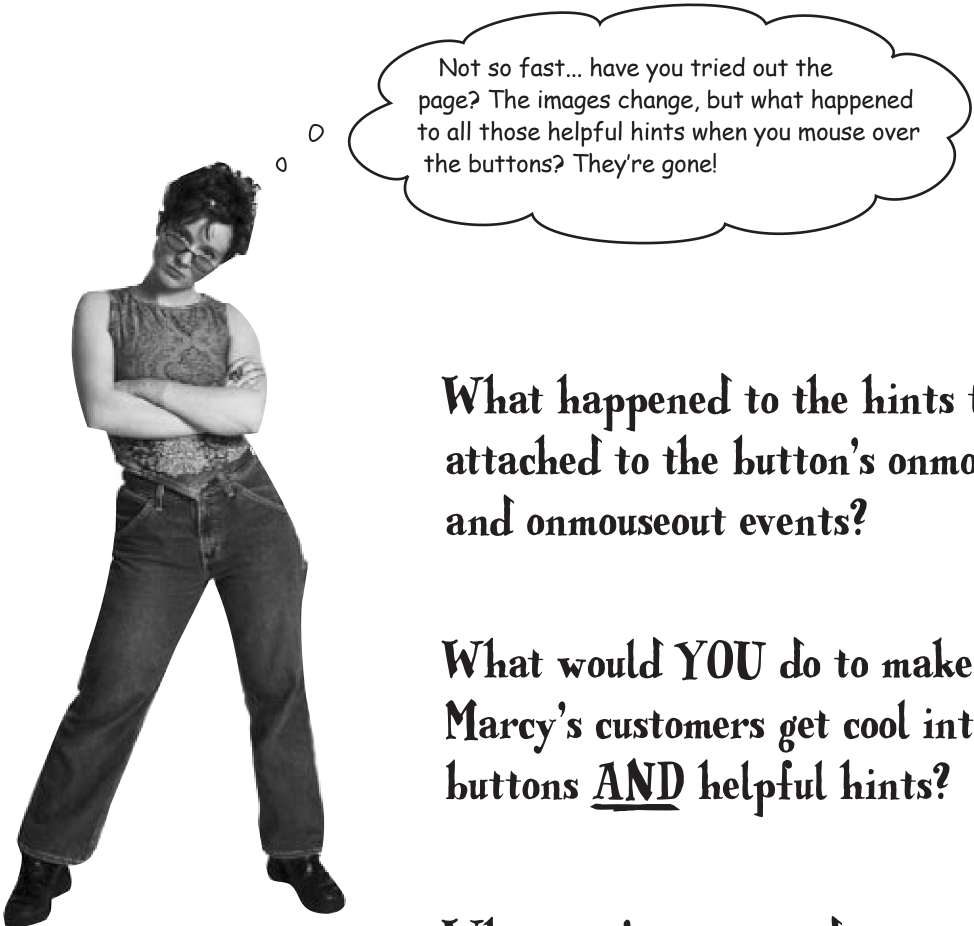
Here are our new event handlers.



(The final) Test Drive

Everything should work! Make all the changes from the last few pages to your XHTML, CSS, and JavaScript, and let's impress Marcy with her stunning new interactive class schedule page.





What happened to the hints that were attached to the button's onmouseover and onmouseout events?

What would YOU do to make sure that Marcy's customers get cool interactive buttons AND helpful hints?

When you've got an idea, turn over to Chapter 4, and let's see how to take your event handling skills to the next level (literally).

4 multiple event handlers

Two's company



A single event handler isn't always enough.

Sometimes you've got more than one event handler that needs to be called by an event. Maybe you've got some event-specific actions, as well as some generic code, and stuffing everything into a single event handler function won't cut it. Or maybe you're just trying to build **clean, reusable code**, and you've got **two bits of functionality** triggered by the **same event**. Fortunately, we can use some **DOM Level 2** methods to assign multiple handler functions to a **single event**.

An event can have only one event handler attached to it (or so it seems)

Marcy's page has a problem. We've assigned two event handlers to the `onmouseover` property of her image buttons:

```
function initPage() {
  // code to deal with tabs

  var buttons =
    document.getElementById("navigation").getElementsByTagName("a");
  for (var i=0; i<buttons.length; i++) {
    var currentBtn = buttons[i];
    currentBtn.onmouseover = showHint;
    currentBtn.onmouseout = hideHint;
    currentBtn.onclick = showTab;
    currentBtn.onmouseover = buttonOver;
    currentBtn.onmouseout = buttonOut;
  }
}
```

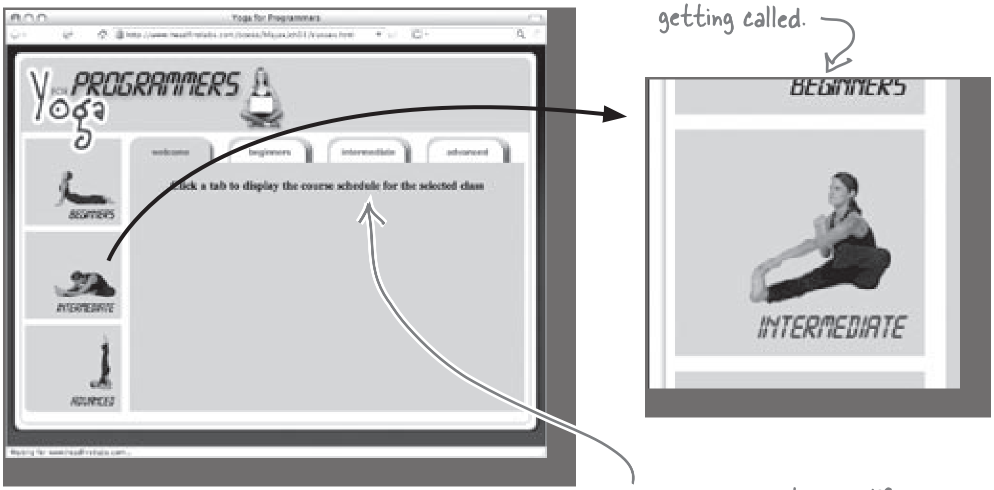
This is the same event: onmouseover for currentBtn. But we're assigning both the showHint() handler...

...and the buttonOver() handler.

Only the LAST event handler assigned gets run

When you assign two event handlers to the same event, only the last event handler that's assigned gets run. So on Marcy's page, mousing over a button triggers `onmouseover`. Then, that event runs the last handler assigned to it: `buttonOver()`.

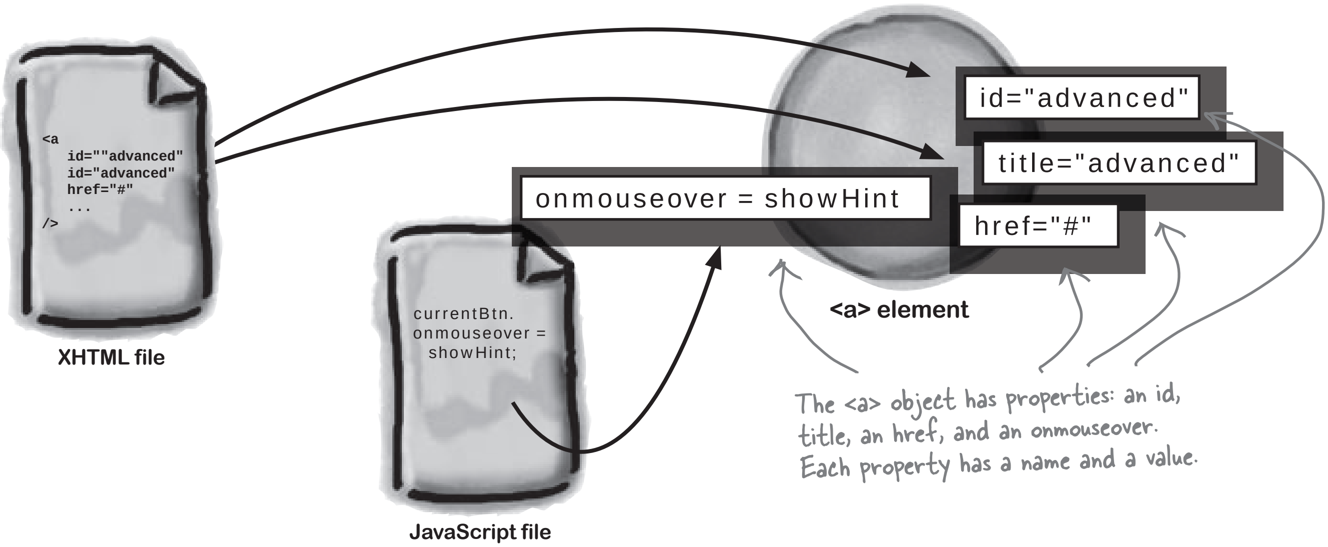
The image is changing when you roll over a button, so buttonOver() is getting called.



But there's no hint anymore. showHint() is *not* getting run.

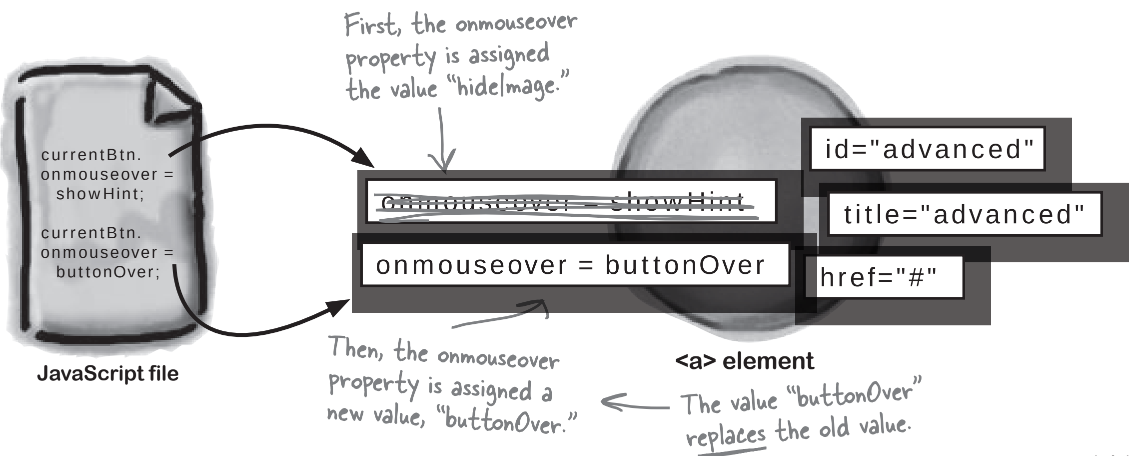
Event handlers are just properties

When you assign an event handler to an event on an XHTML element, the handler becomes a property of the element, just like the `id` or `title` properties of an `<a>` element:



A property can have only ONE value

If you assign a value to a property, that property has that single value. So what happens when you assign another value to that property? The property then has the *new* value, and *the old value is gone*:



Assign multiple event handlers with addEventListener()

So far, we've been adding event handlers to elements by setting the event property directly. That's called the **DOM Level 0** model. DOM stands for the **Document Object Model**, and it's how elements on a web page get turned into objects our JavaScript code can work with.

But DOM Level 0 isn't cutting it anymore. We need a way to assign more than one handler to an event, which means we can't just assign a handler to an event property. That's where **DOM Level 2** comes in. DOM Level 2 gives us a new method, called `addEventListener()`, that lets us assign more than one event handler to an event.

Here's what the `addEventListener()` method looks like:

```
currentBtn.addEventListener("mouseover", showHint, false);
```

Here's the new method. You can call this method on any element you have an object representation for.

For the first argument, use the event name without the "on" in front.

The second argument is the event handler. This should be a function in your script, or you can declare the function inline.

Ignore this for now.

```
currentBtn.addEventListener("mouseover", buttonOver, false);
```

Add a second handler, using addEventListener(), and both handlers will get called for the specified event.

BRAIN POWER

In what order do you think the browser will call your handlers? Do you think the ordering in which the handlers are run will affect how you write your code?

there are no Dumb Questions

Q: DOM? What's that?

A: DOM stands for the Document Object Model. It's a specification that defines how the parts of a web page, like elements and attributes, can be represented as objects that your code can work with.

Q: And what does Level 0 mean?

A: Level 0 was actually an interpretation of the DOM published *before* the DOM was formalized. So it works with the DOM but isn't really part of it.

For your purposes, though, DOM Level 0 is what your browser uses to come up with basic objects and properties for each element in a web page. When you assign a handler to an element's `onmouseover` property, you're using DOM Level 0.

Q: What about DOM Level 1? Do I need to worry about that?

A: Not right now. DOM Level 1 has to do with how you move around in a document. So DOM Level 1 lets you find the parent of an element, or its second child. We'll look at DOM navigation quite a bit in Chapter 6.

Right now, though, you don't really need to worry too much about what level of the DOM you're using, except to make sure your browser supports that level. All major browsers support DOM Level 0 and Level 1, which is why you can assign event handlers programmatically using event properties like `onclick` and `onmouseover`.

Q: And addEventListener() is part of DOM Level 2?

A: Exactly. DOM Level 2 added a lot of specifics about how events should work and dealt with some XML issues that aren't a problem for us right now.

Q: So I can use addEventListener() to add multiple events, and it will work with all the browsers?

A: As long as they support DOM Level 2. But there's one major browser that doesn't support DOM Level 2... we'll look at that in just a minute.

Q: Couldn't I just assign an array to an event property, and give the property multiple values that way?

A: That's a good idea, but it's the browser that connects events to event handlers. If you assigned an array of handler names to an event property, the web browser wouldn't know what to do with that array.

That's why DOM Level 2 was put into place: it provides a standard way for browsers to deal with multiple events. Ideally, specifications standardize a process and remove any possible guesswork.

Q: Why are the event property names different than the names you pass to addEventListener()?

A: That's another great question. That's just the way the authors of the DOM decided to handle event names. So if you're assigning an event property, use `onclick` or `onmouseover`. With `addEventListener()`, use `click` and `mouseover`.

Q: What's that last parameter you're sending to addEventListener()? And why are you setting it to false?

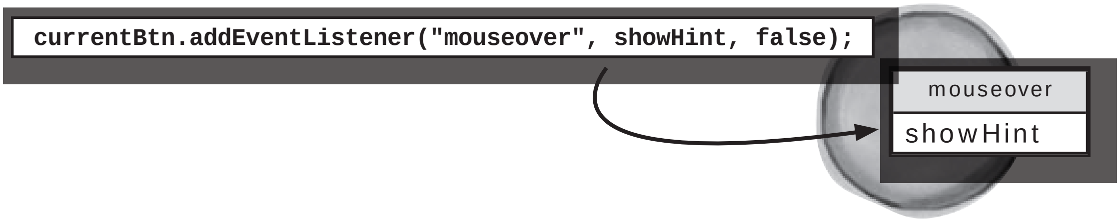
A: That last parameter indicates whether you want event bubbling (`false`) or capturing (`true`). We'll talk more about capturing and bubbling in a bit, so don't worry about this too much. For now, always pass false to `addEventListener()`, which indicates you want event bubbling.

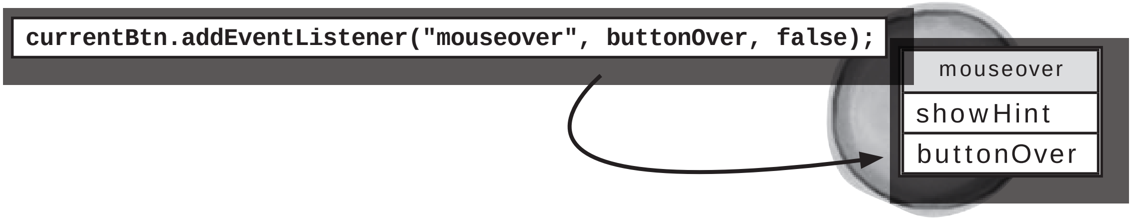
You can assign as many handlers as you want to an event using addEventListener().

addEventListener() works in any web browser that supports DOM Level 2.

Your objects can have multiple event handlers assigned to a single event in DOM Level 2

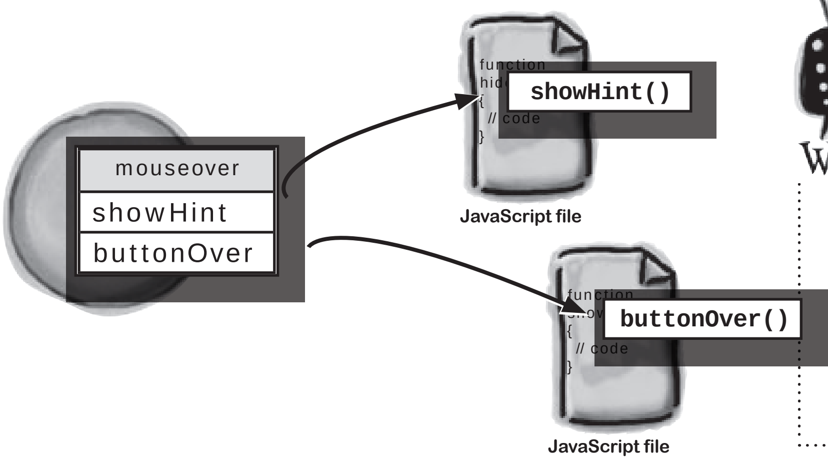
The most important thing that DOM Level 2 added to events is the ability for an event to have more than one handler registered. You've already seen how `addEventListener()` adds a handler to an event:





The browser runs every handler for an event when that event is triggered

When an event is triggered by a mouse movement, the browser looks up the right event. Then, the browser runs every event handler function registered to that event:



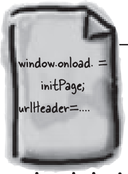
Watch it!

Event listeners aren't called in any particular order.

You might think that the browser calls the event handlers in the order they're added, but that's not guaranteed. Make sure that your handlers don't depend on the order in which they're called.

Sharpen your pencil

It's time to make some improvements to Marcy's yoga page. Below is the current code for initPage(). Your job is to cross out anything that shouldn't be in the code, and make any additions you think you need to get the image button mouse events to work.



schedule.js

```
function initPage() {
  var tabs =
    document.getElementById("tabs").getElementsByTagName("a");
  for (var i=0; i<tabs.length; i++) {
    var currentTab = tabs[i];
    currentTab.onmouseover = showHint;
    currentTab.onmouseout = hideHint;
    currentTab.onclick = showTab;
  }

  var buttons =
    document.getElementById("navigation").getElementsByTagName("a");
  for (var i=0; i<buttons.length; i++) {
    var currentBtn = buttons[i];
    currentBtn.onmouseover = showHint;
    currentBtn.onmouseout = hideHint;
    currentBtn.onclick = showTab;
    currentBtn.onmouseover = buttonOver;
    currentBtn.onmouseout = buttonOut;
  }
}
```

Sharpen your pencil Solution

Your job was to cross out anything that shouldn't be in the code, and make any additions you thought you'd need to get the image button mouse events to work

schedule.js

```
function initPage() {
  var tabs =
    document.getElementById("tabs").getElementsByTagName("a");
  for (var i=0; i<tabs.length; i++) {
    var currentTab = tabs[i];
    currentTab.onmouseover = showHint;
    currentTab.onmouseout = hideHint;
    currentTab.onclick = showTab;
  }

  var buttons =
    document.getElementById("navigation").getElementsByTagName("a");
  for (var i=0; i<buttons.length; i++) {
    var currentBtn = buttons[i];
    currentBtn.onmouseover = showHint;     (crossed out)
     currentBtn.addEventListener("mouseover", showHint, false);
    currentBtn.onmouseout = hideHint;      (crossed out)
     currentBtn.addEventListener("mouseout", hideHint, false);
    currentBtn.onclick = showTab;

    currentBtn.onmouseover = buttonOver;   (crossed out)
     currentBtn.addEventListener("mouseover", buttonOver, false);
    currentBtn.onmouseout = buttonOut;     (crossed out)
     currentBtn.addEventListener("mouseout", buttonOut, false);
  }
}
```

You could have changed these to addEventListener(), but there's really no reason to. They work fine as they are.

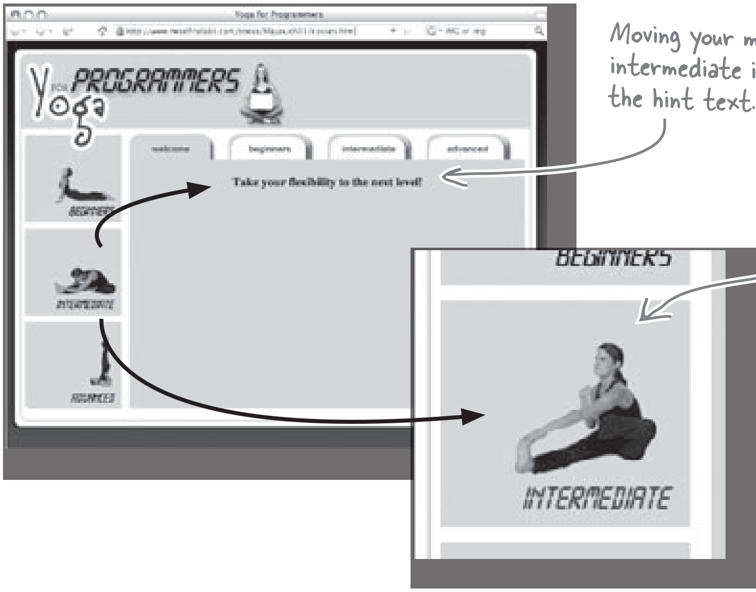
Remember, the event names are the same, but without the "on" before the name.

Both the mouseover and mouseout events need two handlers each.

Use false for each call right now... we'll talk more about bubbling and capturing later.

Test Drive

Change your copy of schedule.js, and fire up your web browser. Try out the image buttons that now use addEventListener(). Does everything work?



Moving your mouse over the intermediate image changes the hint text...

...and updates the button image on the page.

Everything works! Let's have Marcy take a look.



Everything works? Are you kidding? Now the hints don't show up, **and** the images don't change. What's up with that?

What's going on with Marcy's browser?

What do you think is going wrong for Marcy? Why isn't the yoga app working?

....

....

....

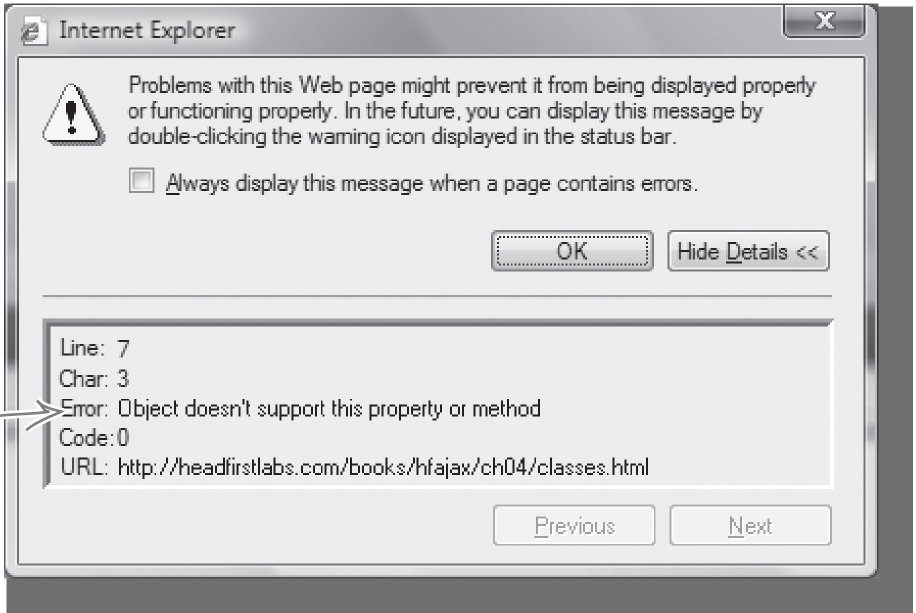
....

Hint: Try different browsers out.

What's going on with Internet Explorer?

The yoga page works great on Firefox, Safari, and a lot of other browsers... but something's definitely wrong on Internet Explorer:

IE shows a little triangular icon in the bottom status bar. Double-click that triangle to see this error message.



There's a problem when you roll over the images. IE reports an error... something about a property or method not being supported?

You can't control what browsers your users are working with.

It's your job to build cross-browser applications... and always test your code in LOTS of browsers.

Internet Explorer uses a totally different event model

Remember that `addEventListener()` only works on browsers that support DOM Level 2? Well, Internet Explorer isn't one of those browsers. IE has its own event model and doesn't support `addEventListener()`. That's why Marcy got an error trying the yoga page out on IE.

Fortunately, IE provides a method that does the same thing as `addEventListener()`. It's called `attachEvent()`:

```
currentBtn.attachEvent("onmouseover", showHint);
```

This is the method that adds the event handler in Internet Explorer.

This time you keep the "on" at the beginning of the event name...

You still give the function the name of the handler to run when the event occurs.

That mysterious "false" disappears in attachEvent().

attachEvent() and addEventListener() are functionally equivalent

Even though the syntax is different, these functions do ***exactly the same thing***. So you just need to use the right one for your users' browsers.

```
currentBtn.attachEvent("onmouseover", showHint);
```

Use attachEvent() for Internet Explorer browsers.

Internet Explorer 7

```
currentBtn.addEventListener("mouseover", showHint, false);
```

Use addEventListener() for Firefox...

...Opera...

...as well as Safari and most other modern browsers.

These are all DOM Level 2 browsers.



The browser wars are just part of web development.

Like it or not, not all browsers are the same. Besides, when Microsoft came up with their own event model, it wasn't that obvious that the DOM Level 2 would take off like it did.

And no matter how it all happened, you can't write off people who use IE... or those who don't.

there are no Dumb Questions

Q: So this is all about which browser is better?

A: No, it's just that not all browsers were developed the same way. It wasn't so long ago that the DOM wasn't a sure thing, and Microsoft just decided to go in another direction. IE isn't better or worse than other browsers; it's just different.

Q: Yeah, but everyone knows IE's a pain. I mean, come on...

A: It's true that lots of web developers think that IE is hard to deal with. That's just because it uses some different syntax. But look at things the other way: if you've been writing code on IE all your life, then it's really Firefox, Safari and Opera that are a pain.

Either way, you've got to write web apps that work on all major browsers, or you're going to miss out on a ton of users.

Q: Why does attachEvent() add the "on" back to the event name?

A: That's just the way IE decided to implement that method.

Q: What about that last argument to addEventListener()? Where did it go on attachEvent()?

A: You may remember that the last argument to `addEventListener()` indicated whether you wanted event bubbling (`false`) or event capturing (`true`). IE only supports event bubbling, so that argument isn't needed.

We'll come back to capturing and bubbling once we've got Marcy's app working on *all* major browsers.

Q: So which one should I use? addEventListener() or attachEvent()?

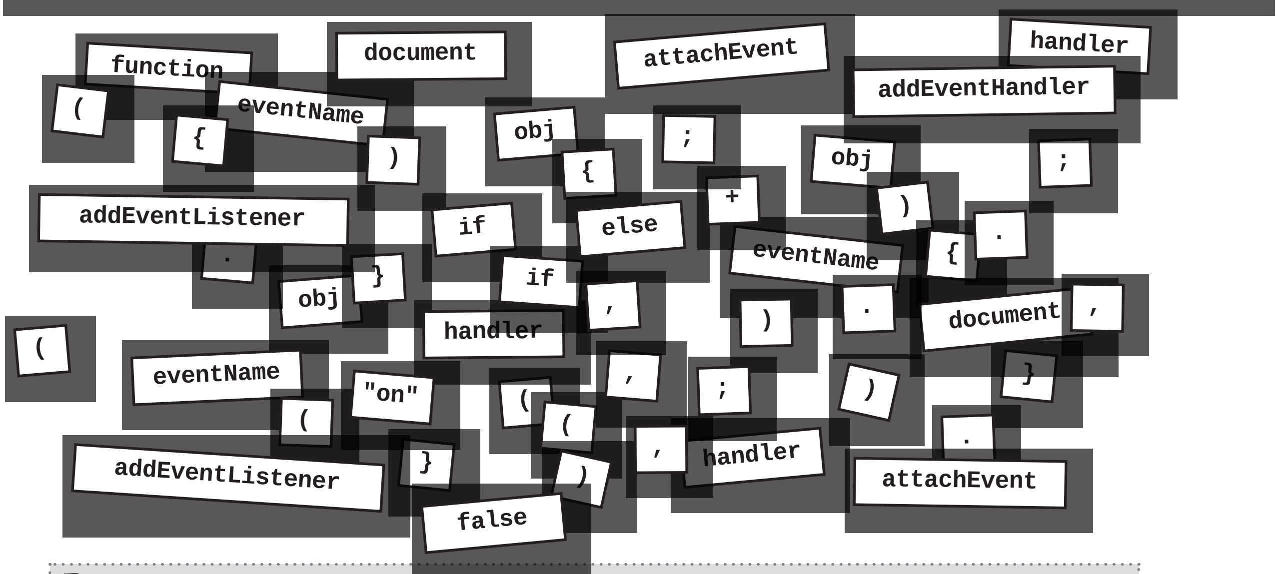
A: Good question. If you think about it and look back at the `createRequest()` function, you probably already know the answer...

In IE, event names have "on" in front, for example, "onclick" and "onmouseover."

In Firefox, Safari, and Opera, event names <u>DON'T</u> have "on" in front: "click" and "mouseover."

Utility Function Magnets

Just like with createRequest(), we need our event handling to work on multiple browsers. Your job is to use the magnets below to build a utility function for adding event handlers to events.



Hint: The expression `(document.someFunction)` returns true if a browser supports running `someFunction()`, and returns false if that function isn't supported.



Utility Function Magnets Solutions

Your job was to figure out a way to get our event handling to run on multiple browsers. You should have built a utility function for adding event handlers to events.

```
function addEventHandler ( obj , eventName , handler ) {
  if ( document . attachEvent ) {
    obj . attachEvent ( "on" + eventName , handler ) ;
  } else if ( document . addEventListener ) {
    obj . addEventListener ( eventName , handler , false ) ;
  }
}
```

This function takes in the object that this event applies to...

...the name of the event, like "click" or "mouseover"...

...and the handler function.

This will only be true for IE.

This is the syntax used by Internet Explorer.

This is the DOM Level 2 syntax.

You could also have checked for addEventListener first... the order of the if/else-if doesn't matter.

there are no Dumb Questions

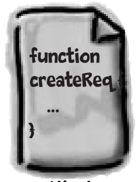
Q: Why didn't we use a try...catch block this time?

A: We could have, but unlike `createRequest()`, this function doesn't need an error to know that something went wrong. Using `document.attachEvent` and `document.addEventListener` works just as well. Besides, the `if...else if` block is a lot easier to read.


Wouldn't it be dreamy if I never had to worry about adding multiple event handlers again? If only I could take my addEventHandler() utility function with me all the time... I guess it's just a fantasy.

addEventHandler() works for ALL apps, not just Marcy's yoga page

So where should you put your code for `addEventHandler()`? We'll use it in Marcy's yoga page, but it's really a utility function. It will work for all our apps and in any browser. So go ahead and add your new code to `utils.js`, so we can reuse it in later web apps we build.



utils.js

```
function createRequest() {
  try {
    request = new XMLHttpRequest();
  } catch (tryMS) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (otherMS) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = null;
      }
    }
  }
  return request;
}
```

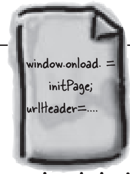
Just like createRequest(), addEventHandler() is useful in all our apps.

```
function addEventHandler(obj, eventName, handler) {
  if (document.attachEvent) {
    obj.attachEvent("on" + eventName, handler);
  } else if (document.addEventListener) {
    obj.addEventListener(eventName, handler, false);
  }
}
```

Anytime you build cross-browser utility functions, store those methods in scripts that you can easily reuse in your other web applications.

Let's update initPage() to use our new utility function

Now we need to change `initPage()`, in `schedule.js`, to use `addEventHandler()` instead of `addEventListener()`. Go ahead and make the following changes to your copy of `schedule.js`:



schedule.js

```
function initPage() {
  var tabs =
    document.getElementById("tabs").getElementsByTagName("a");
  for (var i=0; i<tabs.length; i++) {
    var currentTab = tabs[i];
    currentTab.onmouseover = showHint;
    currentTab.onmouseout = hideHint;
    currentTab.onclick = showTab;
  }

  var buttons =
    document.getElementById("navigation").getElementsByTagName("a");
  for (var i=0; i<buttons.length; i++) {
    var currentBtn = buttons[i];
    addEventHandler(currentBtn, "mouseover", showHint);
    currentBtn.addEventListener("mouseover", showHint, false);
    addEventHandler(currentBtn, "mouseout", hideHint);
    currentBtn.addEventListener("mouseout", hideHint, false);
    currentBtn.onclick = showTab;
    addEventHandler(currentBtn, "mouseover", buttonOver);
    currentBtn.addEventListener("mouseover", buttonOver, false);
    addEventHandler(currentBtn, "mouseout", buttonOut);
    currentBtn.addEventListener("mouseout", buttonOut, false);
  }
}
```

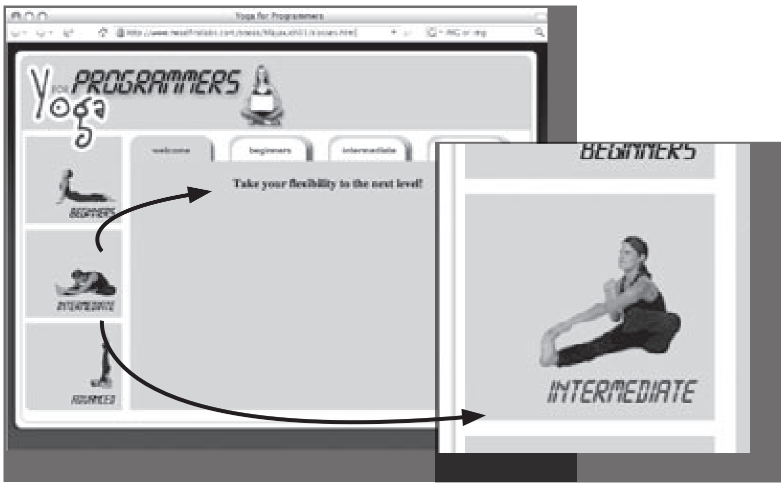
addEventHandler() has to take in the button since it's not a method on that button itself.

Remove all the addEventListener() calls, as they only work on DOM Level 2 browsers.

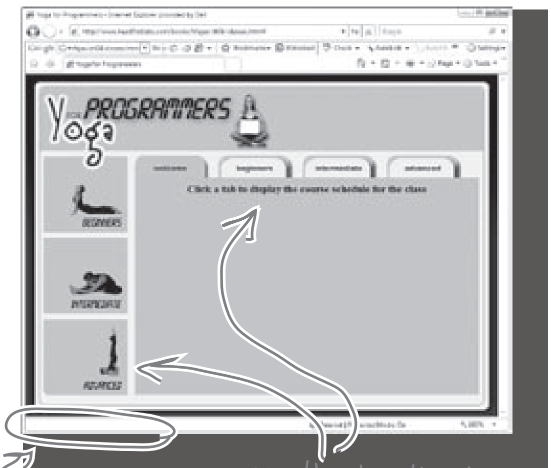
Test Drive

You should have addEventHandler() in utils.js and an updated version of initPage() in schedule.js. Once you've made those changes, try out the yoga page in Internet Explorer and a DOM Level 2 browser, like Firefox or Safari.

Everything works great in DOM Level 2 browsers. That means our addEventHandler() utility function does the right thing for those browsers.



Uh oh... more trouble with IE.



But there's no error reported this time...

...even though neither the hints nor the image rollovers work.

Use an alert() to troubleshoot

Without any error messages, it's hard to know exactly what's going on with Internet Explorer. Try putting in a few `alert()` statements in the event handlers, though, and you'll see they're getting called correctly.

```
function buttonOver() {
  alert("buttonOver() called.");
  this.className = "active";
}

function buttonOut() {
  alert("buttonOut() called.");
  this.className = "";
}
```

So what else could be going wrong?

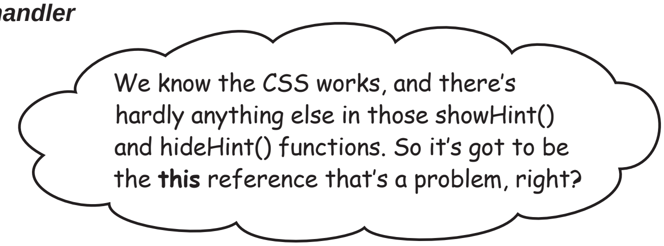
The event handlers are getting called, so that means that `addEventHandler()` is working like it should. And we've already seen that the code in the handlers worked before we added the rollovers. So what else could be the problem?

```
function buttonOver() {
  this.className = "active";
}

function buttonOut() {
  this.className = "";
}
```

We know these class names are right... so what else could be going wrong?

What do you think the problem could be? Can you figure out why the code isn't working like it should?



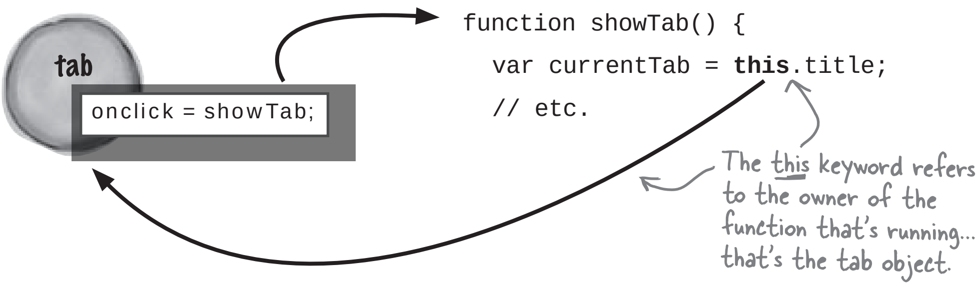


"this" refers to the owner of the executing function.

The `this` keyword in JavaScript always refers to the owner of the function that's executing. So if the method `bark()` was called by an object called `Dog`, then `this` in `bark()` would refer to the `Dog` object.

When you assign an event handler using DOM Level 0, the element with the event is the owner. So if you did `tab.onclick = showTab;`, then in `showTab()`, `this` would refer to the `tab` element. That's why `showHint()` and `hideHint()` worked great in the last chapter.

"owner" and "caller" aren't the same. In a web environment, the browser calls all the functions, but objects representing elements on the page are the owners of those functions.



In DOM Level 2, an event is still the owner of its handlers

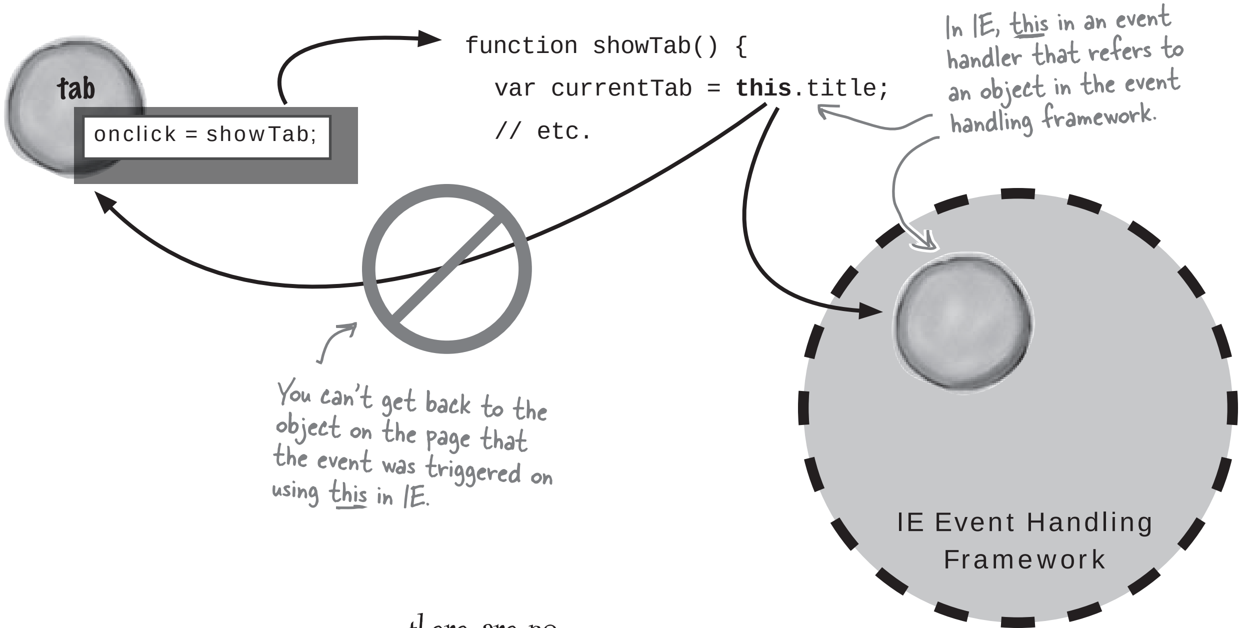
When you're using DOM Level 2 browsers like Firefox, Safari, or Opera, the event handling framework sets the owner of a handler to the object that handler is responding to an event on. So you get the same behavior as with DOM Level 1. That's why our handlers still work with DOM Level 2 browsers.

But what about IE?

Event handlers in IE are owned by IE's event framework, NOT the active page object

You already know that IE doesn't implement DOM Level 2. IE has its own event handling framework. So in IE, ***the event framework owns the handler functions***, not the object on the XHTML page that was activated with a click or mouse over. In other words, `this` in `showTab()` refers to the IE event framework, *not* to a `tab` element on Marcy's yoga web page.

Framework just means a set of objects or code that performs some task, like handling events on a web page.



there are no Dumb Questions

Q: What object does *this* refer to in the IE framework, then?

A: `this` *always* refers to the owner of the function that's currently running. So in an event handler under IE's event framework, `this` points at one of the framework's objects.

It really doesn't matter what that object is because it's not all that useful. What we need is a way to get information about the element that the event occurred on.

Q: But if this is how IE handles events, how did our code work back in Chapter 3 on Internet Explorer?

A: Our code worked in IE because we were just using DOM Level 0 syntax. Anytime you assign a handler to a property, like `currentBtn.onmouseover = showTab`, that's DOM Level 0.

But our code now is using `addEventListener()` and `attachEvent()`. That's *not* DOM Level 0, and now `this` doesn't mean the same thing as it did in that earlier code.

Q: Ok, great. So the page still doesn't work in Internet Explorer. What now?

A: Well, take a moment to think about what exactly you need. It's not so much the `this` keyword that's important, but the information that keyword let us access.

What exactly do we need to know about in our event handler functions?

attachEvent() and addEventListener() supply another argument to our handlers

One of the cooler things about JavaScript is that you don't need to list all the arguments that your functions take when you declare that function. So even if your function declaration is `showTab()`, you can pass arguments to `showTab()` when you call it.

```
function showTab() {
  var currentTab = this.title;
  // etc.
```

Even though there aren't any objects listed here, showTab() could still be getting additional information when it's called.

We've got to replace "this" with something that points to the object on the web page that triggered this event.

The bad thing about that is sometimes you miss out on arguments that are passed to your function.

Your event handlers get an Event object from attachEvent() and addEventListener()

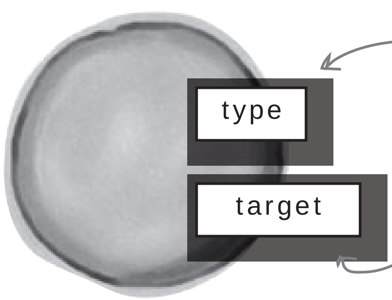
When you register an event handler using DOM Level 2 and `addEventListener()`, or `attachEvent()` and IE, both frameworks pass your event handlers an object of the `Event` type.

Your handlers can then use this object to figure out what object on a page was activated by an event, and which actual event was triggered.

There are two properties in particular that are really helpful to know about. The first is `type`, which gives the name of the event that was triggered, like "mouseover" or "click." The second is `target`, which gives you the target of the event: the object on the page that was activated.

Event objects know what object triggered them and what type of event they are.

So we need to get access to the Event object in our handler functions.



This is the name of the event that occurred. It's the same string that you passed to addEventListener(), like "mouseover" or "onload."

This is the object the event occurred on, like a tab or an image on a web page.

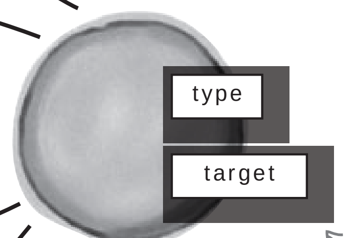
The target of an Event object is equivalent to what you get from the "this" keyword in DOM Level 2 browsers.

We need to name the Event argument, so our handlers can work with it

You don't have to list all the arguments a JavaScript function gets. But if you want to actually use those arguments in the function, you ***do*** need to list the arguments. First, we need to get access to the `Event` object in our handlers, so we can figure out what object on a page triggered a call to our handler. Then, we need to list the argument for that `Event` object:

```
function showHint(e) {
  // function code
}
function hideHint(e) {
  // function code
}
function showTab(e) {
  // function code
}
function buttonOver(e) {
  // function code
}
function buttonOut(e) {
  // function code
}
```

All of the event handlers in schedule.js should now take an extra argument.



Event object

DOM Level 2 browsers and IE send an Event object to a handler when it's triggered.

We'll need to change most of our handler functions to use the Event object instead of "this."

BRAIN POWER

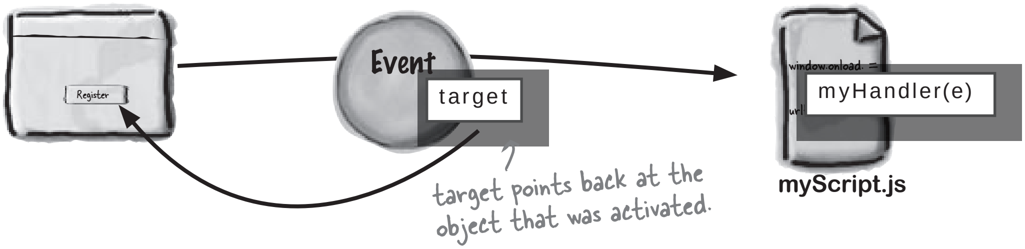
With DOM Level 2 browsers, you can use either `this` or the `Event` object passed into an event handler to find out what element was activated. Do you think one approach is better than the other? Why?

You say ~~tomato~~ target, I say ~~tomato~~ srcElement...

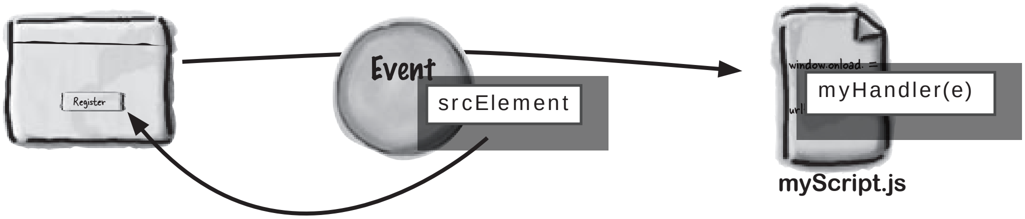
The good news is that both IE and DOM Level 2 browsers make the object that triggered an event available. The bad news is that DOM Level 2 and IE use different versions of the `Event` object, each with different properties.

In some cases, the `Event` object properties refer to the same thing, but the property *names* are different. And to make matters worse, modern versions of IE pass in an `Event` object, but earlier versions of IE make the `Event` object available as a property of the `window` object.

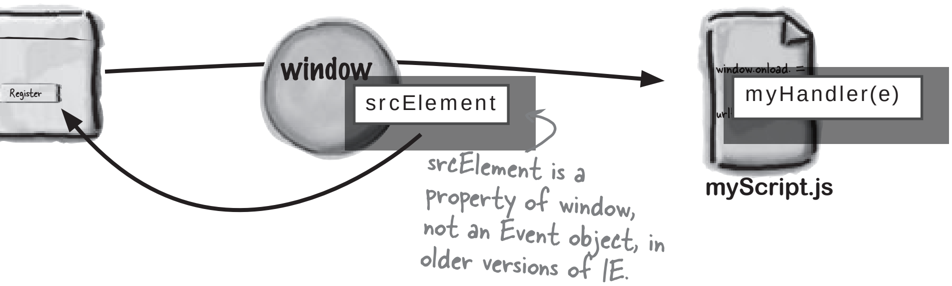
Browsers that support DOM Level 2, like Firefox, Safari, and Opera, pass an Event object to event handlers. The Event object has a property named “target” that refers to the object that triggered the event.



Internet Explorer 7 passes an Event object to event handlers. The Event object has a property named “srcElement” that refers to the object that triggered the event.



Earlier versions of Internet Explorer provide the object that triggered an event in a property named “srcElement,” available on the window object.



WHO DOES WHAT?

So you think you really know your browsers? Here's a quiz to help you check your knowledge out for real. For each property, method, or behavior on the left, check off all the boxes for the browsers that support that thing. Good luck!

	Firefox	IE 7	Safari	Opera	IE 5
addEventListener()					
srcElement					
DOM Level 2					
target					
addEventHandler()					
var currentTab = this.title;					
DOM Level 0					
window.srcElement					
attachEvent()					

Who Does What?

So you think you really know your browsers? Here's a quiz to help you check your knowledge out for real. For each property, method, or behavior on the left, you were to check off all the boxes for the browsers that support that thing.

	Firefox	IE 7	Safari	Opera	IE 5
addEventListener()	✓		✓	✓	
srcElement		✓			✓
DOM Level 2	✓		✓	✓	
target	✓		✓	✓	
addEventHandler()	✓	✓	✓	✓	✓
var currentTab = this.title;	✓	?	✓	✓	?
DOM Level 0	✓	✓	✓	✓	✓
window.srcElement					✓
attachEvent()		✓			✓

addEventListener(): This is a DOM Level 2 function. No IE.

srcElement: All versions of IE have this property, but on different objects.

target: Only DOM Level 2 browsers support target.

addEventHandler(): This is our utility function, so it works on all browsers.

var currentTab = this.title;: this is tricky. It works in all browsers with DOM Level 0 events, but not in IE if attachEvent() is used.

window.srcElement: Old versions of IE expose srcElement as a property of the window object.

there are no Dumb Questions

Q: So "this" always refers to the function that called my function?

A: No, `this` refers to the *owner* of the function. Sometimes that's another bit of code, but it also might be an object, like a tab on a form that got clicked.

Q: But that's not true in Internet Explorer, right?

A: `this` still refers to the owner of a function in IE. The difference is that when you use `attachEvent()`, the owner of your function is an object in IE's event handling framework and not an object on your web page.

The DOM provides an object-based model of your web page that your code can work with.

getElementById(), the document object, and the onclick property are all aspects of using the DOM in your code.

Q: So we shouldn't ever use "this" in Internet Explorer?

A: Actually, `this` is still a very useful part of JavaScript, whether you're using IE or a DOM Level 2 browser. But if you're writing an event handler function, you're probably better off avoiding `this`. If you're writing an event handler that's going to be called using the IE event handling framework, via `attachEvent()`, then you've *got* to avoid using `this`.

Q: I'm still a little fuzzy on all this DOM stuff. Can you explain that again?

A: DOM, or the Document Object Model, is how a browser represents your page as objects. JavaScript uses the DOM to work with a web page. So every time you change an element's property or get an element with `getElementById()`, you're using the DOM. That's all you really need to know right now, but we're going to dig into the DOM a lot more in just a few chapters.

Q: As long as I use addEventHandler(), I don't have to worry about all this DOM stuff, though, right?

A: Well, you don't have to worry about whether you should use `attachEvent()` or `addEventListener()`. But as you'll see in Chapter 6, there's still a lot of DOM work you'll end up doing.

`addEventHandler()` takes care of registering an event handler to an event in a browser-neutral way. In other words, `addEventHandler()` works with all modern browsers.

Q: And that's why it's in utils.js, right? Because it's a utility function?

A: Right. `addEventHandler()` works for all browsers and many different kinds of applications, not just Marcy's yoga page. So it's best put into a reusable script, like `utils.js`.

Q: But even if we use addEventHandler(), we've still got these issues with target and srcElement, right?

A: Right. IE 7 passes off to event handlers an `Event` object with a `srcElement` property that points at the object that triggered the event. Older versions of IE make that same object available through the `window.srcElement` property. DOM Level 2 browsers provide an `Event` object with a property called `target` pointing to the object that triggered an event.

Q: I've heard that object called an "activated object" before. Is that the same thing?

A: Yes. An **activated object** just means an object that represents an element on a web page that an event occurred on. So if an image is clicked, the JavaScript object that represents that image is the activated object.

Q: Since addEventHandler() took care of adding events on all browsers, why don't we just build another utility function to deal with all this target/srcElement stuff?

A: Now *that* is a great idea!

So how do we actually GET the object that triggered the event?

The best way to deal with differences in how IE and DOM Level 2 browsers handle events is another utility function. Our handler functions are now getting `Event` objects, but what we really need is the **activated object**: the object representation of the element on the page that the event occurred on.

So let's build a utility function to take the event argument we get from those browsers, and figure out and return the activated object:

```
function getActivatedObject(e) {
  var obj;
  if (!e) {
    // early version of IE
    obj = window.event.srcElement;
  } else if (e.srcElement) {
    // IE 7 or later
    obj = e.srcElement;
  } else {
    // DOM Level 2 browser
    obj = e.target;
  }
  return obj;
}
```

Our handlers get an Event object, so let's pass that object to this utility function.

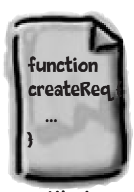
Early versions of IE actually don't send an object...

...which tells us to check the srcElement property of the window object.

IE 7 has a srcElement property, which is what we want on that browser.

DOM Level 2 browsers provide the activated object in the target property of the passed-in event.

This function goes in utils.js along with createRequest() and addEventHandler().



utils.js

Exercise

You need to update Marcy's code again. In all of the event handlers, you need to use getActivatedObject() to get the activated object. You'll also need to change the rest of those methods to use the object returned from that function instead of `this`. There are a few other changes you should already have made, too. Check off each task once you're finished.

Update utils.js

Add the `addEventHandler()` and `getActivatedObject()` functions to the file.

- [ ] `addEventHandler()`
- [ ] `getActivatedObject()`

Use addEventHandler() instead of addEventListener()

Use the generic `addEventHandler()` to abstract out DOM Level 2 and IE event handling differences.

- [ ] Update `initPage()` to only use `addEventHandler()`

Use getObject() instead of this

Update all your event handler functions to use `getActivatedObject()` instead of the `this` keyword. You'll need to make other changes to get those functions working as well.

- [ ] `showHint()`
- [ ] `hideHint()`
- [ ] `buttonOver()`
- [ ] `buttonOut()`
- [ ] `showTab()`

When you think you're done, try things out for yourself. Then, turn the page to see how we updated the code in schedule.js and utils.js.

Exercise Solution

Your job was to complete the changes to schedule.js so that all the event handlers would take an Event argument, and use the getObject() utility function from utils.js to figure out the activated object. You should have also removed all references to `this` in your event handler functions.

```
window.onload = initPage;
var welcomePaneShowing = true;

function initPage() {
  var tabs =
    document.getElementById("tabs").getElementsByTagName("a");
  for (var i=0; i<tabs.length; i++) {
    var currentTab = tabs[i];
    currentTab.onmouseover = showHint;
    currentTab.onmouseout = hideHint;
    currentTab.onclick = showTab;
  }

  var buttons =
    document.getElementById("navigation").getElementsByTagName("a");
  for (var i=0; i<buttons.length; i++) {
    var currentBtn = buttons[i];
    addEventHandler(currentBtn, "mouseover", showHint);
    addEventHandler(currentBtn, "mouseout", hideHint);
    currentBtn.onclick = showTab;
    addEventHandler(currentBtn, "mouseover", buttonOver);
    addEventHandler(currentBtn, "mouseout", buttonOut);
  }
}

function showHint(e) {
  if (!welcomePaneShowing) {
    return;
  }
  var me = getActivatedObject(e);
  switch (me.title) {
    case "beginners":
      var hintText = "Just getting started? Come join us!";
      break;
    case "intermediate":
      var hintText = "Take your flexibility to the next level!";
      break;
    case "advanced":
      var hintText = "Perfectly join your body and mind " +
                     "with these intensive workouts.";
      break;
    default:
```

Since these events have just a single handler, DOM Level 0 is fine.

You probably did this step earlier. All the multiple event handling situations should now be setup with addEventHandler().

Make sure you add the extra argument to all your event handler functions, so you can work with the object sent to those handlers.



schedule.js

```
    var hintText = "Click a tab to display the course " +
                   "schedule for the class";
  }
  var contentPane = document.getElementById("content");
  contentPane.innerHTML = "<h3>" + hintText + "</h3>";
}

function hideHint(e) {
  if (welcomePaneShowing) {
    var contentPane = document.getElementById("content");
    contentPane.innerHTML =
      "<h3>Click a tab to display the course schedule for the class</h3>";
  }
}

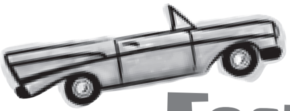
function showTab(e) {
  var selectedTab = this.title;
  var me = getActivatedObject(e);
  var selectedTab = me.title;
  if (selectedTab == "welcome") {
    welcomePaneShowing = true;
    document.getElementById("content").innerHTML =
      "<h3>Click a tab to display the course schedule for the class</h3>";
  } else {
    welcomePaneShowing = false;
  }

  // everything else is the same...
}

function buttonOver(e) {
  var me = getObject(e);
  me.classNameActivated = "active";
 this.className = "active";
}
function buttonOut(e) {
  var me = getActivatedObject(e);
  me.className = "";
  this.className = "";
}
```

It's common to call the object returned from getObject() "me."

The "me" variable stands in for "this." The code stays almost exactly the same.

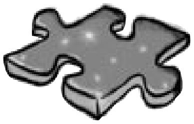


Test Drive

It's been a long journey, but you're finally ready to test out Marcy's yoga page one more time. See if everything works in both IE browsers AND DOM Level 2 browsers.







EventAcrostic

Take some time to sit back and give your right brain something to do. Answer the questions in the top, then use the letters to fill in the secret message where the numbers match.

This model uses object.event = handler syntax

1 2 3 4 5 6 7 8 9

Use this function to register an event in DOM Level 2

10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

This is what Marcy teaches

26 27 28 29

Use this function to register an event in Internet Explorer

30 31 32 33 34 35 36 37 38 39 40

This is the object that triggered the event

41 42 43 44 45 46

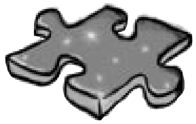
This event happens when the user presses a key

47 48 49 50 51 52 53 54 55

22 6 13 39 31 35 10 55 1 18 19 16 28

19 20 32 35 5 49 15 26 17 2

19 23 40 7 25 29 34 21 19 37 19 41 51



EventAcrostic

Did you figure out the secret message? Do you agree with it?

This model uses object.event = handler syntax

D	O	M	L	E	V	E	L	O
1	2	3	4	5	6	7	8	9

Use this function to register an event in DOM Level 2

A	D	D	E	V	E	N	T	L	I	S	T	E	N	E	R
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

This is what Marcy teaches

Y	O	G	A
26	27	28	29

Use this function to register an event in Internet Explorer

A	T	T	A	C	H	E	V	E	N	T
30	31	32	33	34	35	36	37	38	39	40

This is the object that triggered the event

T	A	R	G	E	T
41	42	43	44	45	46

This event happens when the user presses a key

O	N	K	E	Y	D	O	W	N
47	48	49	50	51	52	53	54	55

E	V	E	N	T		H	A	N	D	L	I	N	G
22	6	13	39	31		35	10	55	1	18	19	16	28

I	S		T	H	E		K	E	Y		T	O
19	20		32	35	5		49	15	26		17	2

I	N	T	E	R	A	C	T	I	V	I	T	Y
19	23	40	7	25	29	34	21	19	37	19	41	51

5 asynchronous applications

It's like renewing your driver's license



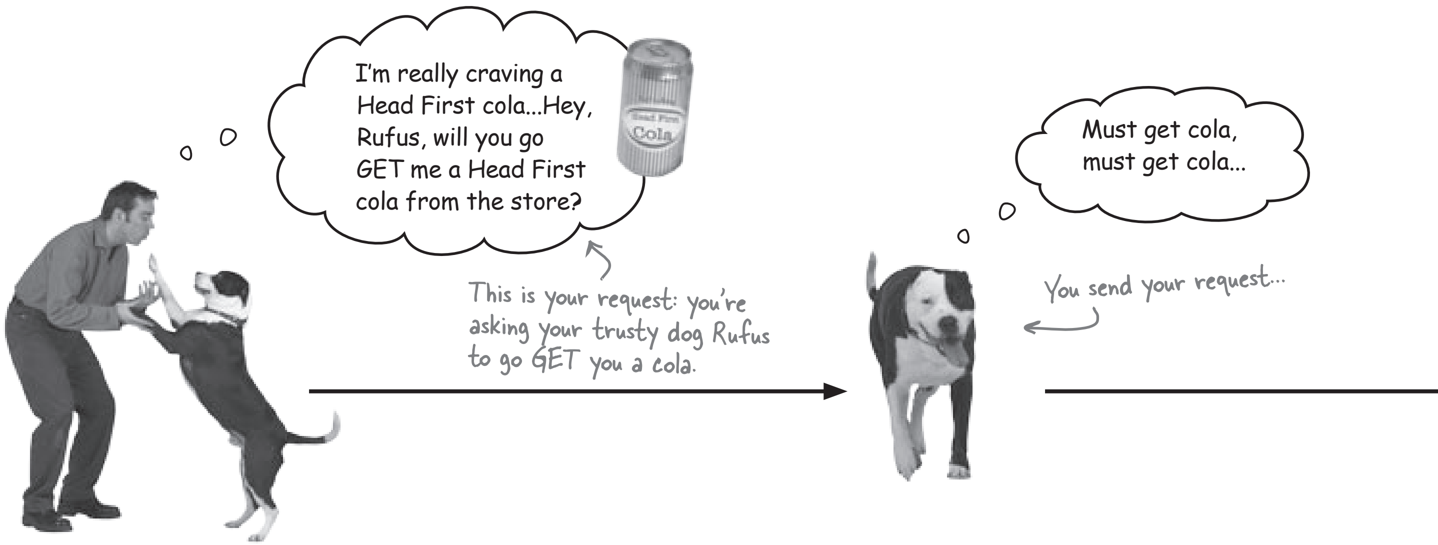
Are you tired of waiting around? Do you hate long delays? You can do something about it with asynchrony!

You've already built a couple of pages that made asynchronous requests to the server to avoid making the user sit around waiting for a page refresh. In this chapter, we'll dive even deeper into the details of building asynchronous applications. You'll find out what **asynchronous really means**, learn how to use **multiple asynchronous requests**, and even build a **monitor function** to keep all that asynchrony from confusing you and your users.

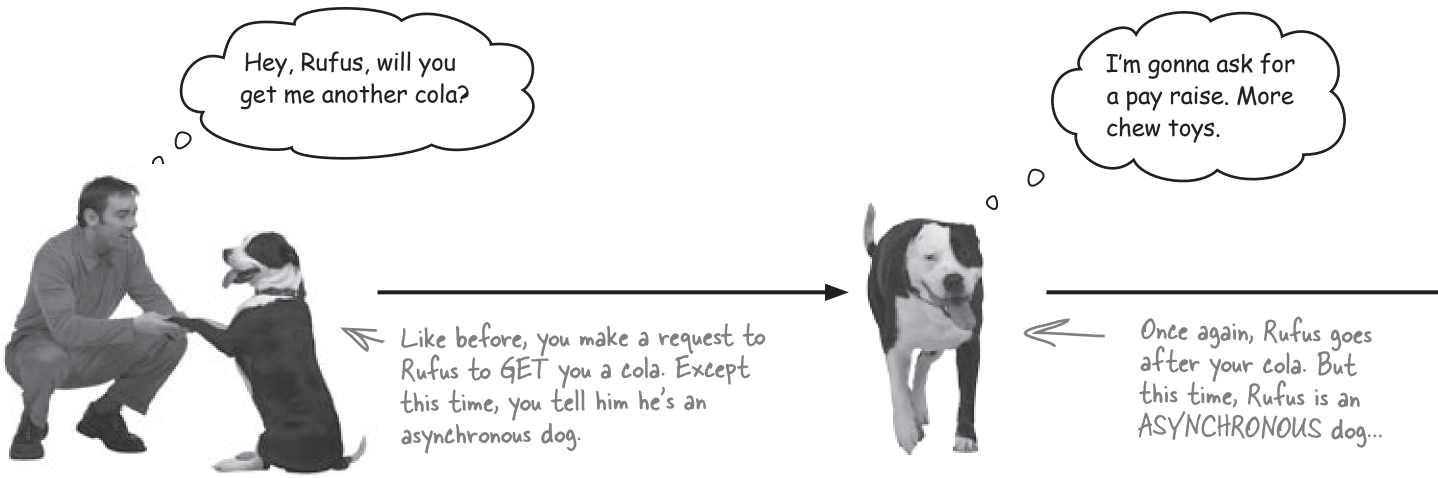
What does asynchronous really mean?

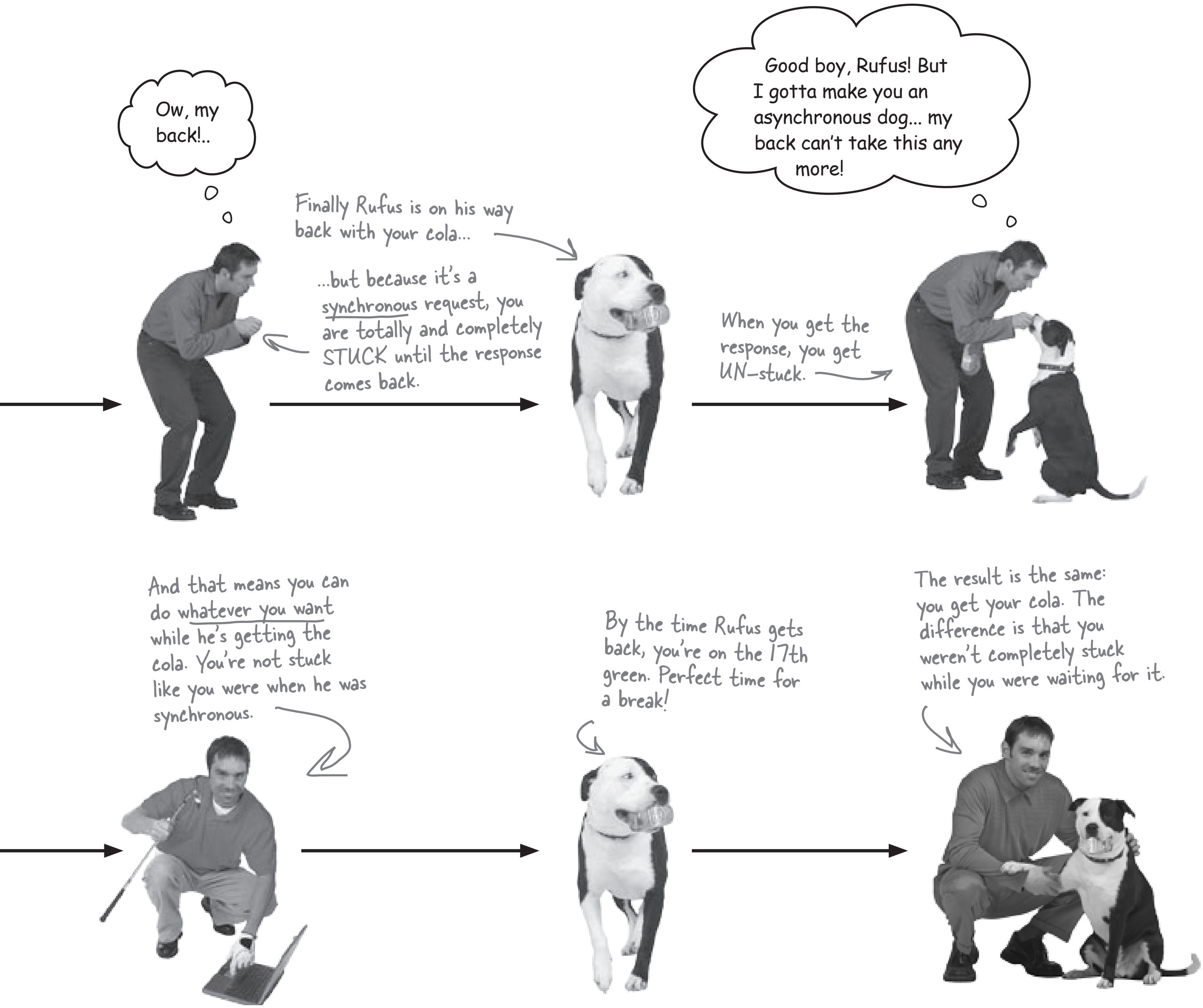
An **asynchronous request** means that you don't have to **wait around** while a web server is responding to that request. That means you're not stuck: you can go on doing what you want, and have the server let you know when it's finished with your request. Let's take a view of this from 10,000 feet by first looking at what a synchronous request is, and then comparing it to an asynchronous request:

A synchronous request for cola



An asynchronous request for cola




Ow, my back!..
Finally Rufus is on his way back with your cola...
...but because it's a synchronous request, you are totally and completely STUCK until the response comes back.
When you get the response, you get UN-stuck.
Good boy, Rufus! But I gotta make you an asynchronous dog... my back can't take this any more!
And that means you can do whatever you want while he's getting the cola. You're not stuck like you were when he was synchronous.
By the time Rufus gets back, you're on the 17th green. Perfect time for a break!
The result is the same: you get your cola. The difference is that you weren't completely stuck while you were waiting for it.

You've been building asynchronous apps all along

Take a look back at the app you built for Mike's Movies in Chapter 2. When a user types in their username and leaves that field, that value gets sent to the server right away for validation. But the user can fill out the rest of the form while that validation is happening. That's because we made the request to the server **asynchronous**.

checkUsername() gets called when a user leaves the username field.

```
function checkUsername() {
  document.getElementById("username").className = "thinking";
  request = createRequest();
  if (request == null)
    alert("Unable to create request");
  else {
    var theName = document.getElementById("username").value;
    var username = escape(theName);
    var url= "checkName.php?username=" + username;
    request.onreadystatechange = showUsernameStatus;
    request.open("GET", url, true);
    request.send(null);
  }
}
```

Remember this last argument to request.open()? It says, "Make this an asynchronous request. Don't make the user wait on the server's response."

Here's where we send a request to the server.

Validate Username



The server sends back response data, like a readyState, status code, and eventually the response itself.

response data

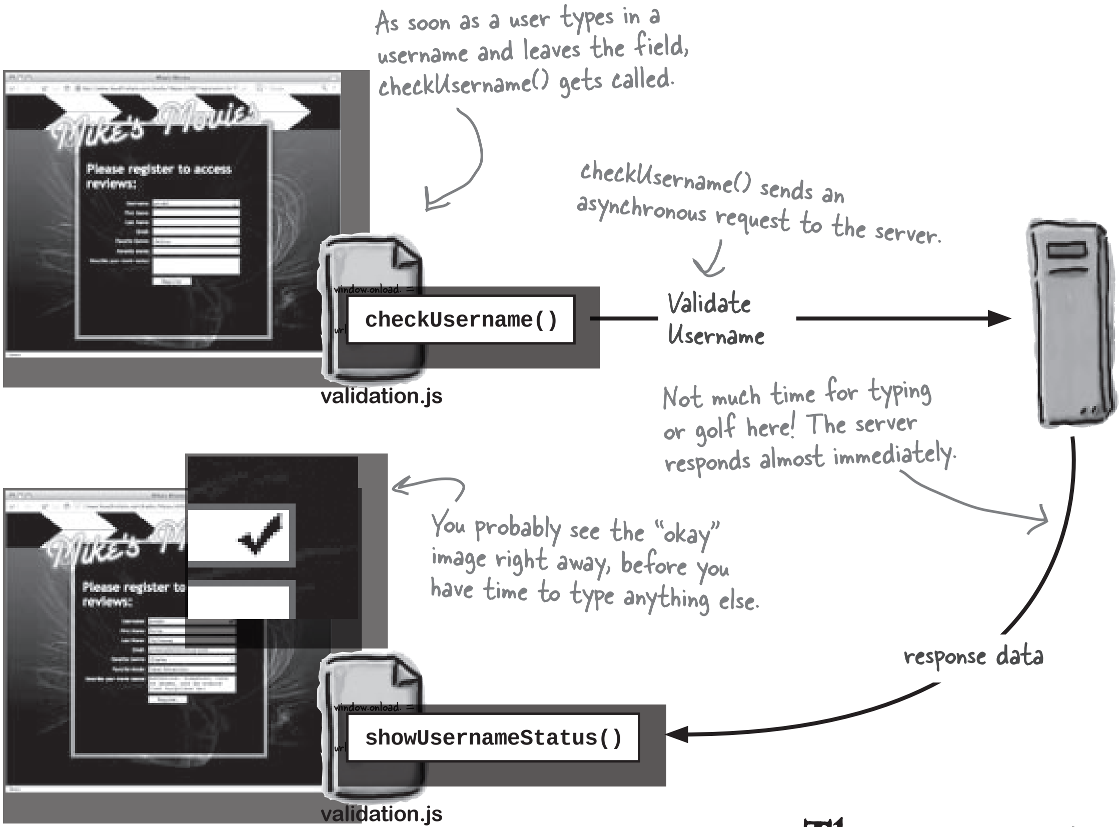
```
function showUsernameStatus() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      if (request.responseText == "okay") {
        document.getElementById("username").className = "approved";
        document.getElementById("register").disabled = false;
      } else {
        document.getElementById("username").className = "denied";
        document.getElementById("username").focus();
        document.getElementById("username").select();
        document.getElementById("register").disabled = true;
      }
    }
  }
}
```

This callback runs every time the server responds with new data.

All of the server processing, AND the callback, happens while the user is still filling out the form... no waiting around.

But sometimes you barely even notice...

When you built Mike's Movies, you probably barely noticed the asynchrony. Requests to and from a server—especially when you're developing, and there's not a lot of network traffic—hardly take any time at all.



But the response time on a live site is almost always going to be ***slower***. There are more people competing for server resources, and user machines and connections may not be as powerful and fast as your development machine. And that doesn't even take into account how long it takes a server to respond. If the server's querying a huge database, or has to do lots of server-side processing, that slows the request and response cycle down, too.

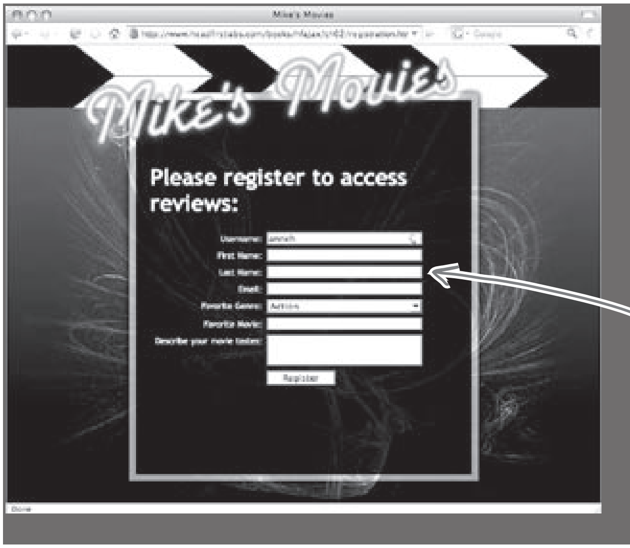
The response time on a live site will almost always be slower than on a test site.

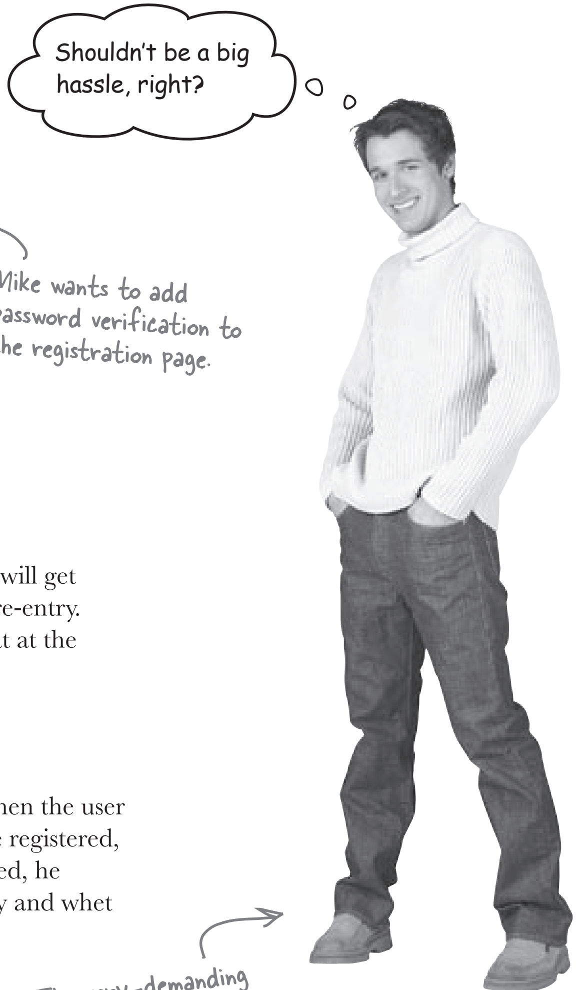
The only way to know for sure is to TEST your app on the live site.

Speaking of more server-side processing...

Mike loves the page you built him and has some more ideas. His site's become popular, but some folks have been posting fake reviews under other peoples' usernames. Mike needs you to add a password field to the registration form, verify the username *and* password asynchronously, and keep unwanted users out of his system for good.

Mike's got a server-side program that checks a password to make sure it's at least 6 characters long and contains at least one letter. That should be good enough for his movie site.





Oh, and password verification...

Mike actually wants *two* password fields. The first password value will get sent to the server for validation. The second field is for password re-entry. The value in both password fields have to match. We'll handle that at the client's browser.

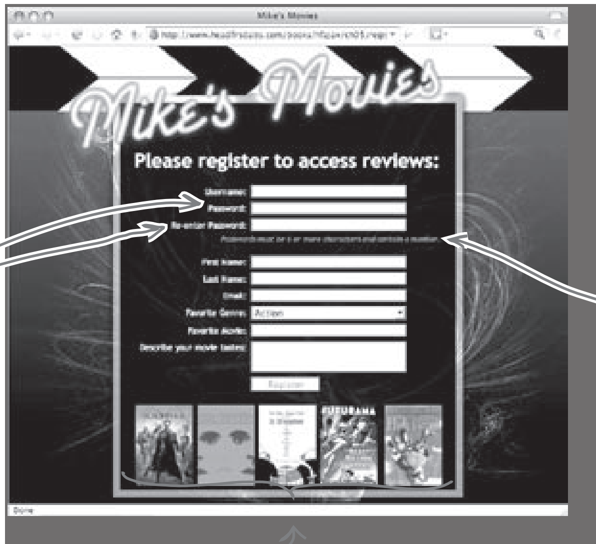
And... how about some eye candy, too?

Mike's not happy with how long it takes to get a user processed when the user clicks the Register button. Since we can't let users in until they are registered, Mike's got an idea: while the form is being submitted and processed, he wants images from his collection of movies and posters to scroll by and whet the user's appetite for reviews on those items.

Sharpen your pencil

There's a lot to do on Mike's site. Below, we've given you a screenshot of what Mike's final app should look like. It's up to you to make notes about the interactions that need to happen to add all the behavior Mike's asking for.

What should be happening between the page and your JavaScript, and your JavaScript and the server?



There are two new password fields. We'll need to add those to Mike's XHTML.

Mike also lays out his password requirements in red text. That's just more XHTML.

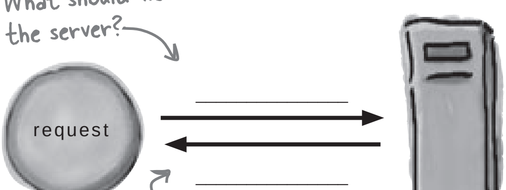
Mike wants images of his reviewed movies at the bottom of the page.



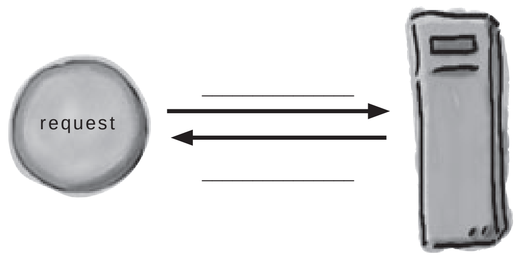
web server

Don't forget about the server-side requirements! You'll need two different server-side processes for this version of Mike's app. Label the arrows below to indicate what you're sending to the server, and what you think it should send back in response.

What should we send to the server?

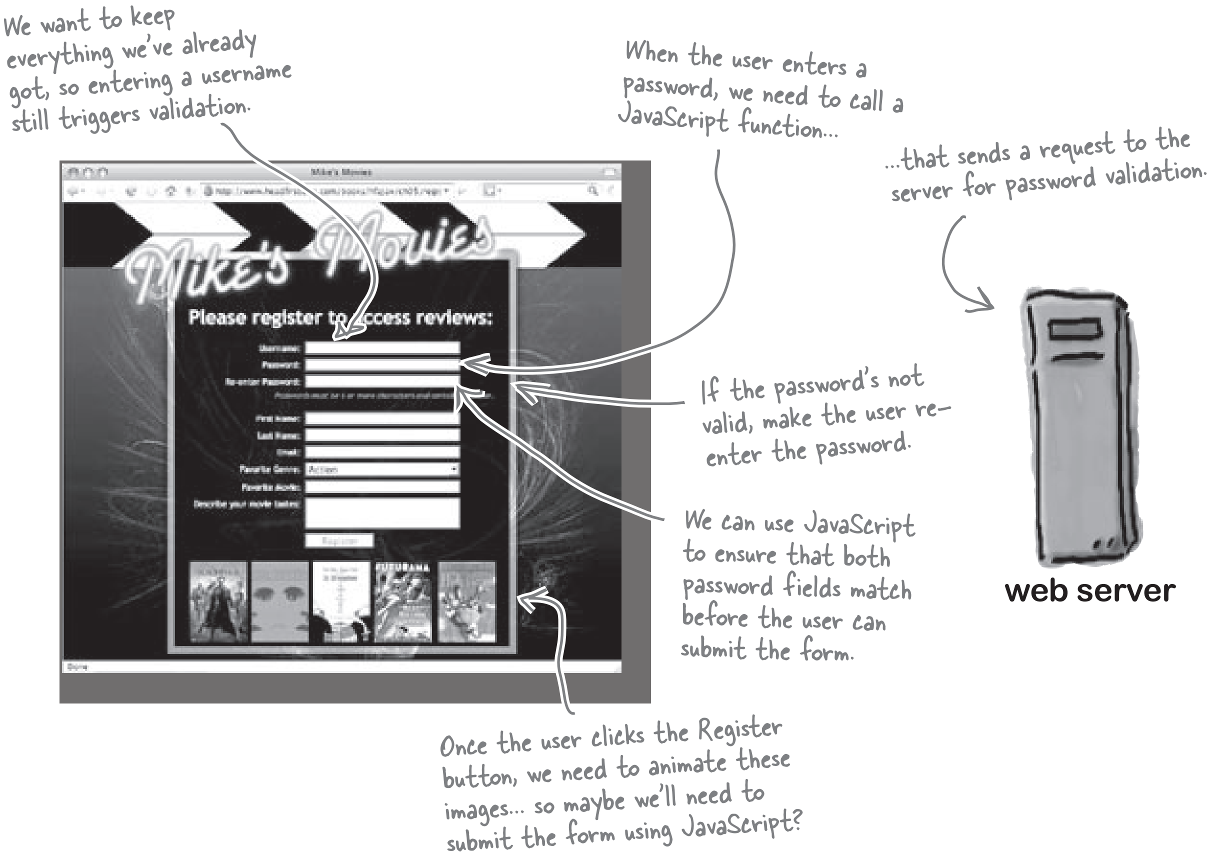


What should the server send back?

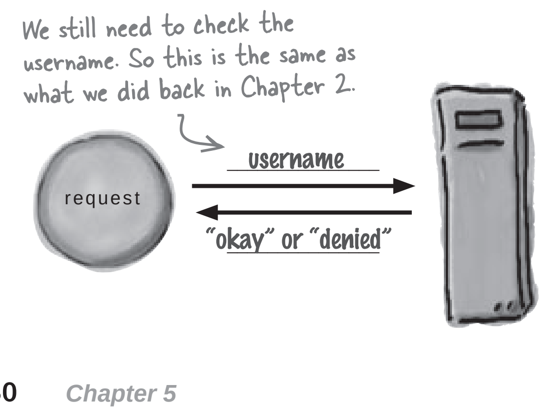


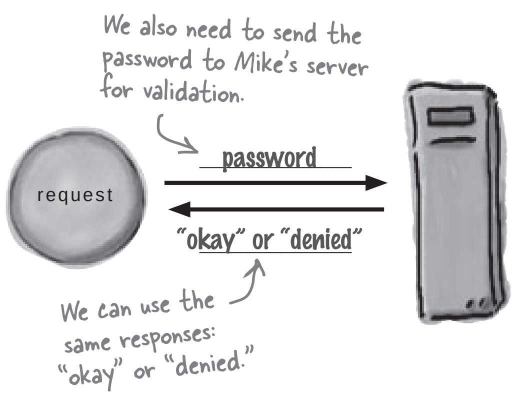
Sharpen your pencil Solution

There's a lot to do on Mike's site. Your job was to figure out the interactions that have to occur to get all this behavior working like it should.



Were you able to figure out what our JavaScript needed to send to Mike's server, and what the server-side programs should send back?





(More) Asynchrony in 3 easy steps

We need to finish up Mike's web page, and then add all the extra interactions that he wants. Then, we've got to figure out a way to submit his form and animate those images at the bottom.

Here's how we're going to take on the improved version of Mike's Movies in this chapter:

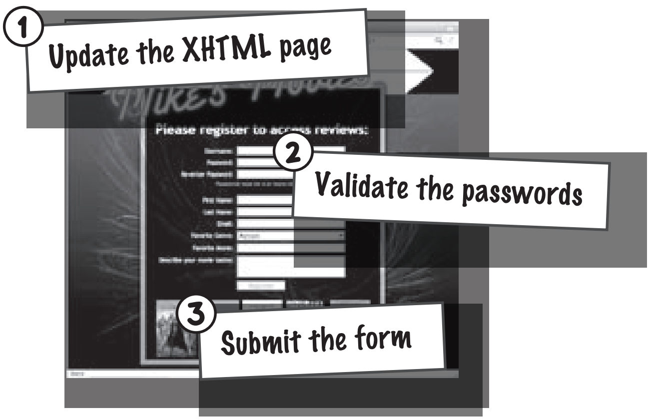
1 Update the XHTML page
We need to add two more password fields: one for entering a password, and one for verifying that password. We'll also need a section for putting in those movie images.

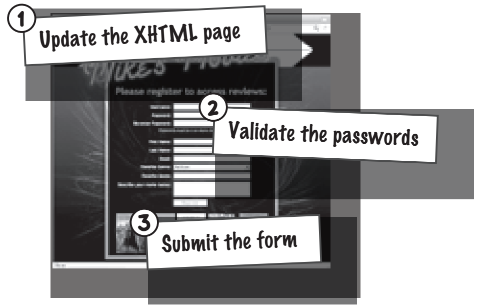
2 Validate the user's passwords
Then, we need to handle the user's password. We've got to build a handler function that takes a password, sends it to the server, and sets up a callback that checks to see if the password was valid. Then we can use the same icons we used on the username field to let the user know if their password is valid.

Somewhere in here we should make sure both password fields match, too.

3 Submit the form
Finally, we've got to build code to submit the form, and animate the images along the bottom. We can attach that code to the Register button's `click` event instead of letting the form submit through a normal XHTML Submit button.

We need code to submit the form since we've got to animate the images at the same time.





We need two password fields and a <div> for the cover images

We've got to add a couple of password fields to the form, and then we also need a `<div>` at the bottom to hold all those cover images. Here are the changes you should make to your copy of `registration.html`:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Mike's Movies</title>
  <link rel="stylesheet" href="css/movies.css" />
  <script src="scripts/utils.js" type="text/javascript"></script>
  <script src="scripts/validation.js" type="text/javascript"></script>
</head>
<body>
<div id="wrapper">
  <h1>Please register to access reviews:</h1>
  <form action="register.php" method="POST">
    <ul>
      <li><label for="username">Username:</label><input id="username"
                 type="text" name="username" /></li>
      <li><label for="password1">Password:</label><input id="password1"
                 type="password" name="password1" /></li>
      <li><label for="password2">Re-enter Password:</label><input id="password2"
                 type="password" name="password2" /></li>
      <li class="tip">Passwords must be 6 or more characters and
                      contain a number.</li>
      <li><label for="firstname">First Name:</label><input id="firstname"
                 type="text" name="firstname" /></li>
      <li><label for="lastname">Last Name:</label><input id="lastname"
                 type="text" name="lastname" /></li>
      <li><label for="email">Email:</label><input id="email"
                 type="text" name="email" /></li>
      <li>
        <label for="genre">Favorite Genre:</label>
        <select name="genre" id="genre">
          <option value="Action">Action</option>
          <option value="Comedy">Comedy</option>
          <option value="Crime”>Crime</option>
          <option value="Documentary">Documentary</option>
          <option value="Drama">Drama</option>
          <option value="Horror">Horror</option>
          <option value="Musical">Musical</option>
          <option value="Romance">Romance</option>
          <option value="SciFi">Sci-Fi/Fantasy</option>
```

We're using the same scripts as before. We can just add new code to validation.js.

We need two fields: one for the initial password, and one to verify the password.

Make the type of these "password" so nobody can see what users are typing.

This label lays out Mike's password requirements, and the CSS styles it to be red.

```
          <option value="Suspense">Suspense</option>
          <option value="Western">Western</option>
        </select>
      </li>
      <li><label for="favorite">Favorite Movie:</label><input id="favorite"
                 type="text" name="favorite" /></li>
      <li><label for="tastes">Describe your movie tastes:</label><textarea
                 name="tastes" cols="60" rows="2" id="tastes"></textarea></li>
      <li><label for="register"></label><input id="register"
                 type="submit" value="Register" name="register" /></li>
    </ul>
  </form>
  <div id="coverBar">
    <img src="images/coverMatrix.jpg" width="82" height="115"
         style="left: 0px;" />
    <img src="images/coverDeadRingers.jpg" width="82" height="115"
         style="left: 88px;" />
    <img src="images/coverDrStrangelove.jpg" width="82" height="115"
         style="left: 176px;" />
    <img src="images/coverFuturama.jpg" width="82" height="115"
         style="left: 264px;" />
    <img src="images/coverHolyGrail.jpg" width="82" height="115"
         style="left: 356px;" />
    <img src="images/coverRaisingArizona.jpg" width="82" height="115"
         style="left: 444px;" />
    <img src="images/coverRobotChicken.jpg" width="82" height="115"
         style="left: 532px;" />
  </div>
</div>
</body>
</html>
```

This is pretty straightforward. We add a <div> with an id...

...and then a bunch of movie covers that Mike said he's got reviews for.



Download the CSS and graphics from Head First Labs.

Go to the Head First Labs site and download the examples for Chapter 5. You'll find the cover graphics, as well as a version of `registration.html` that matches this XHTML, and a new version of `movies.css` to go with the new XHTML.

there are no Dumb Questions

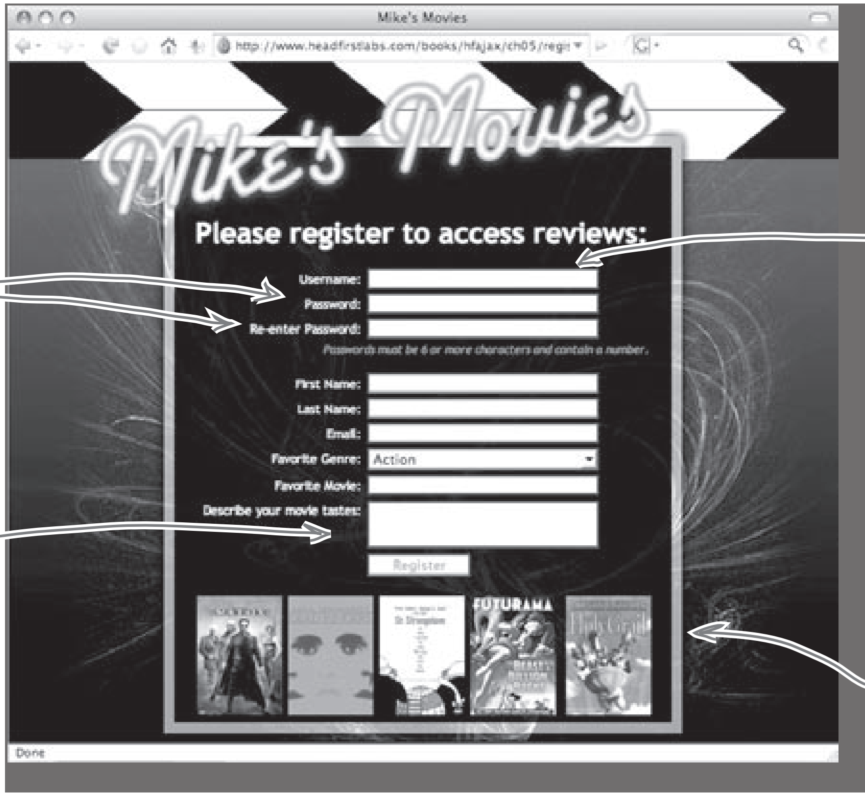
Q: Why are you using style attributes on those cover images? Isn't mixing style into the XHTML a really bad idea?

A: It is. But the only other option is to have a different class for each image in that `<div>`. It's good to try and separate content from presentation, but if it makes your XHTML and CSS a real mess, then you sometimes have to break a rule to make your XHTML and CSS manageable. Who wants to keep up with 10 or 15 different CSS classes, one for each movie image?

Test Drive

Check out Mike's Movies... with password and images.

Once you've made all the changes to registration.html, or downloaded the examples, open up the page in your web browser. Make sure that all the cover images show up, and that there are two password fields. You should also check that the username field still sends a request to the server for validation, and that the Register button is disabled when the page first loads.



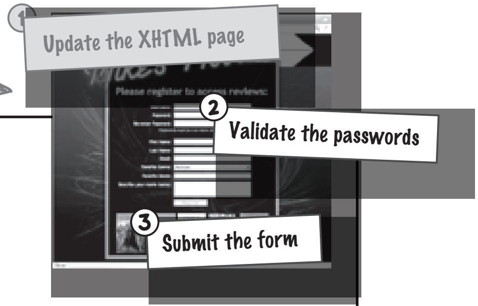
There should be two password fields. Also make sure that only asterisks appear when you type in these fields.

The username field should still work. You should be able to type in a username, get an In Progress icon, and then either a check mark or an X.

The Register button should be disabled right now.

Here are all of Mike's movie images... we can animate and scroll these later.

With the XHTML done, we can move on to validating passwords.





Procedure Magnets

By now, you should be pretty comfortable figuring out how to tie an event on a page to a request for a server-side program to process some data. Put the magnets under the right task. Order doesn't matter in most cases, so just match the magnet to what that magnet helps you accomplish.

To handle an event:

To send a request object to the server:

Register each handler function to the correct event(s).

Create a callback function that will be called when the server responds to the request

Obtain an instance of a request object

Trigger the event registration function before the user can work with the web page.

Send the request object

Configure the request object

Create a function to register handlers to events.

Write event handlers for each event you want to perform behavior on.



Procedure Magnet Solution

Your job was to build a process for connecting an event on a web page to a server-side program.

To handle an event:

Here's where your event-specific behavior occurs. Nothing works without event handlers.

- **Write event handlers for each event you want to perform behavior on.**
- **Create a function to register handlers to events.**
- **Register each handler function to the correct event(s).**
- **Trigger the event registration function before the user can work with the web page.**

You can create these in any order. You just have to have all four things in place before your code will work.

We've been calling this initPage().

Obtain a reference to an object, and then either assign the handler to its event property or use addEventHandler() to register the handler to an event on that object.

The statement "window.onload = initPage()" makes sure event handlers are set up before users can work with a page.

To send a request object to the server:

These have to happen in this specific order.

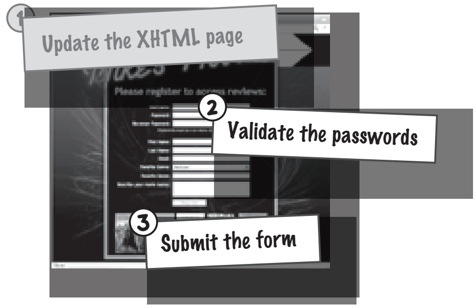
- **Obtain an instance of a request object**
- **Configure the request object**
- **Send the request object**
- **Create a callback function that will be called when the server responds to the request**

createRequest() in utils.js handles this task.

You need to give the request a URL to send information to, and a callback for the browser to call when the server responds.

You use request.send() for this.

This should take the server's response and do something with that response.



If you need new behavior, you probably need a new event handler function

We've got to validate a password once a user enters something in the password form fields. So we need a new event handler to validate passwords. We also need to register an `onblur` event handler for the right password field.

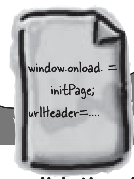
`validation.js` already sets up event handlers in `initPage()`, so we just need to add a new event handler assignment:

```
window.onload = initPage;

function initPage() {
  document.getElementById('username').onblur = checkUsername;
  document.getElementById('password2').onblur = checkPassword;
  document.getElementById('register').disabled = true;
}

function checkPassword() {
  // We'll write this code next
}
```

All we need to do is assign another event handler, this time to the password2 field.



validation.js

there are no Dumb Questions

Q: Why didn't you use addEventHandler() to register the checkPassword() handler?

A: Because we're only assigning one handler to the password2 field. If we needed multiple handlers for that field, then you would need DOM Level 2 or IE's `attachEvent()`. In those cases, you'd want to use `addEventHandler()`. But since this is a single handler on an event, we can stick with DOM Level 0.

Sharpen your pencil

Why do you think checkPassword() is registered to the password2 field, and not the password1 field?

....

....

....

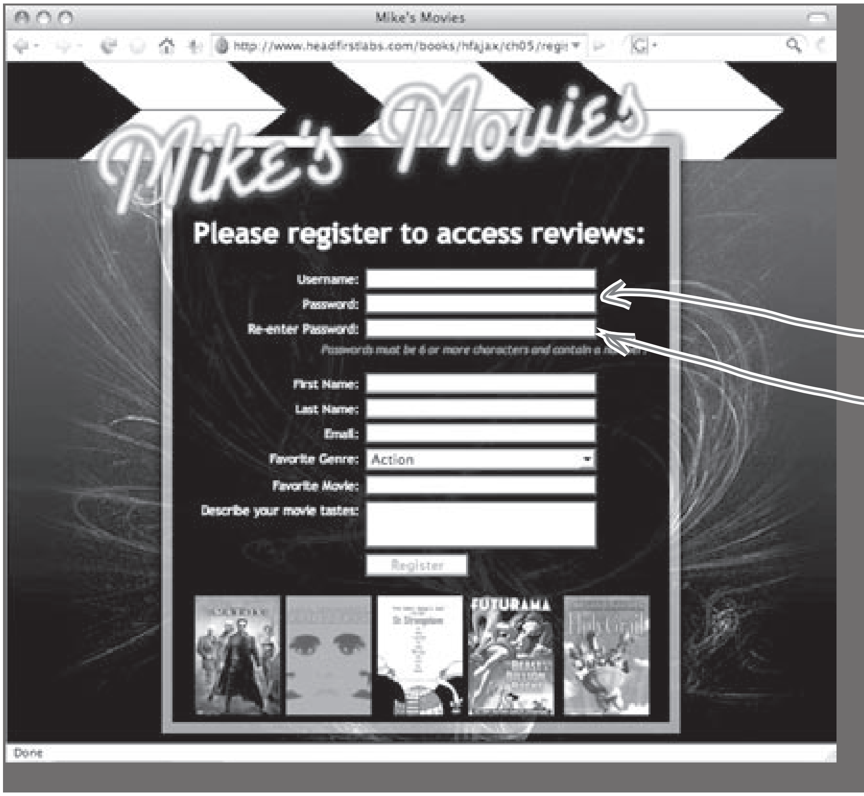
....

Sharpen your pencil Solution

Why do you think checkPassword() is registered to the password2 field, and not the password1 field?

We need to check the passwords against each other before sending them to the server for validation. So we can't do anything until the user's entered a password for both password fields.

Your answer doesn't have to be exactly the same, but it should be pretty close.



When there's a value for the first password field, we could send a request to the server...

...but then what do we do if the second password field doesn't match? Our request would be meaningless.

We really need to check and see if both fields match first, and then send the password to the server for validation.

Exercise

It's time to write some code. Using what you've already figured out, plus the hints below, you should be able to write the code for the checkPassword() event handler and the showPasswordStatus() callback. Take your time... you can do it.

A callback runs when the server returns a response to your request. An event handler runs when a certain event on your page occurs.

Hints:

- **There's a CSS class called "thinking" that you can set either password field to in order to get an "in progress" icon. The "approved" class shows a check mark, and the "denied" class shows an X.**

- **The program on the server that validates passwords is at the URL "checkPass.php". The program takes a password and returns "okay" if the password is valid, and "denied" if it's not. The parameter name to the program should be "password."**

Write the code for checkPassword() and a callback called showPasswordStatus() here:

Exercise Solution

Your job was to write the code for the checkPassword() event handler and the showPasswordStatus() callback. See how close your solution is to ours.

```
function checkPassword() {
  var password1 = document.getElementById("password1");
  var password2 = document.getElementById("password2");
  password1.className = "thinking";

  // First compare the two passwords
  if ((password1.value == "") || (password1.value != password2.value)) {
    password1.className = "denied";
    return;
  }

  // Passwords match, so send request to server
  var request = createRequest();
  if (request == null) {
    alert("Unable to create request");
  } else {
    var password = escape(password1.value);
    var url = "checkPass.php?password=" + password;
    request.onreadystatechange = showPasswordStatus;
    request.open("GET", url, true);
    request.send(null);
  }
}
```

Since we'll use these field elements a lot, it makes sense to put them both into variables.

As soon as we start, we need to show the "in progress" icon.

First, make sure the password1 field isn't empty.

Then, we need to compare the values of the two fields.

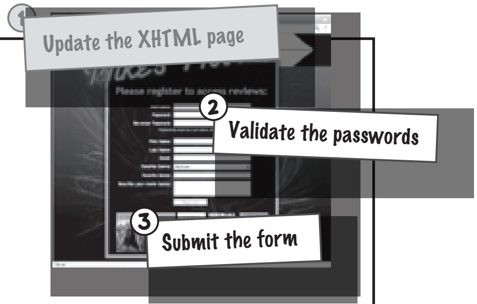
If the non-empty passwords don't match, show an error and stop processing.

This is pretty standard. Get a request object, and make sure it's good to use.

We can use either password field's value... we know they're the same now.

Set the callback.

We're making this an asynchronous request. That will be really important later...



Make sure this function name exactly matches the value of the onreadystatechange property of the request object.

```
function showPasswordStatus() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var password1 = document.getElementById("password1");
      if (request.responseText == "okay") {
       password1.className = "approved";
       document.getElementById("register").disabled = false;
      } else {
       password1.className = "denied";
       password1.focus();
       password1.select();
       document.getElementById("register").disabled = true;
      }
    }
  }
}
```

If we get a response of "okay", show the check mark icon for the password1 field.

If the password's not valid, change the CSS class...

Remember to enable the Register button!

...move to the password1 field...

...and highlight the password1 field.

Since the password isn't valid, we can't let the user register, so disable that button.

there are no Dumb Questions

Q: Should we be sending a password as part of a GET request? Is that safe?

A: Great question! We'll talk a lot more about GET, and how secure it is, in Chapter 12. For now, just focus on the details of asynchrony, and we'll look at securing Mike's users' passwords a bit better later on.

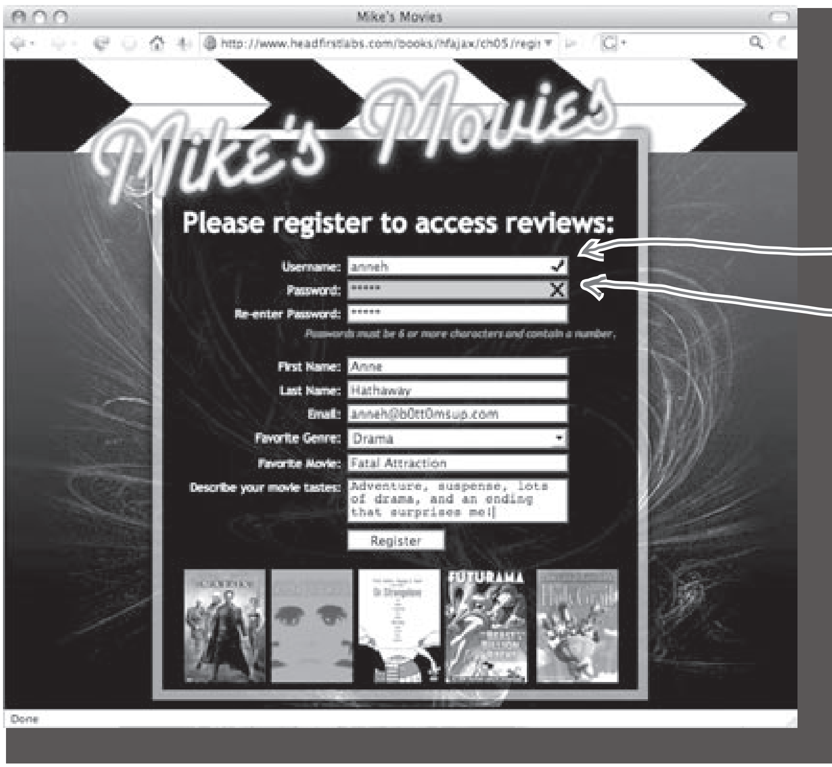
Q: I tried this out, and I think there are some problems...

A: Really? What were they? What do you think caused them? Try out our code, and see what you get. Are there things you would change or improve? Try entering in just a username, or just valid passwords. What do you see happening?

Test Drive

How does Mike's page look and behave?

Make the changes to `validation.js` that we did, or use your own version (as long as it does the same basic things). Then, try the page out. What's happening? Do you think our code works, or are there problems?

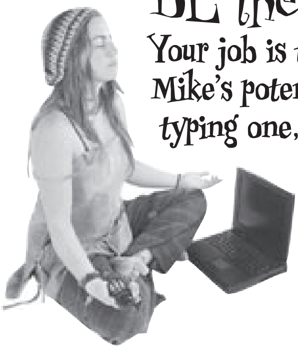


The username field still works... that's good.

Hmmm.... the first password field has an X. What exactly does that mean? Is it totally clear to users? We may have to come back to that a little later.

BE the USER

really fast (annotation pointing to "the")



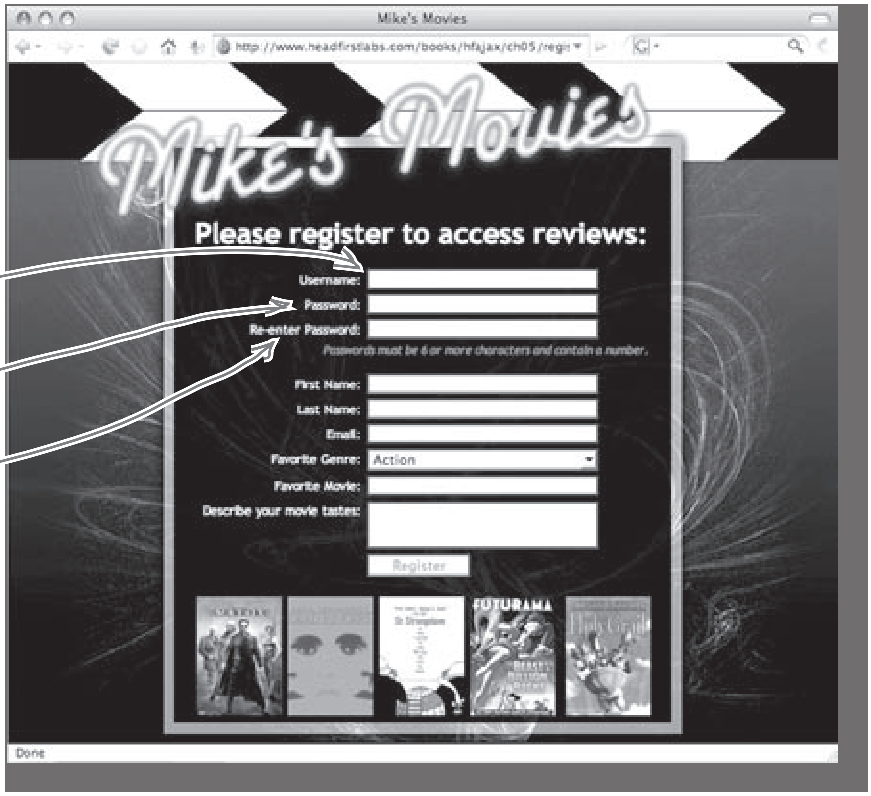
Your job is to play like you're one of Mike's potential customers... and a fast-typing one, at that. Try and figure out what happens when someone types in a username, and then quickly moves to typing a password in both of the password fields.

Type in a username...

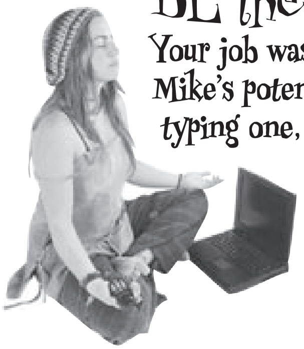
...and then quickly type in your password, once...

...and then again.

Finally, tab out of the second password field to trigger the checkPassword() event handler.



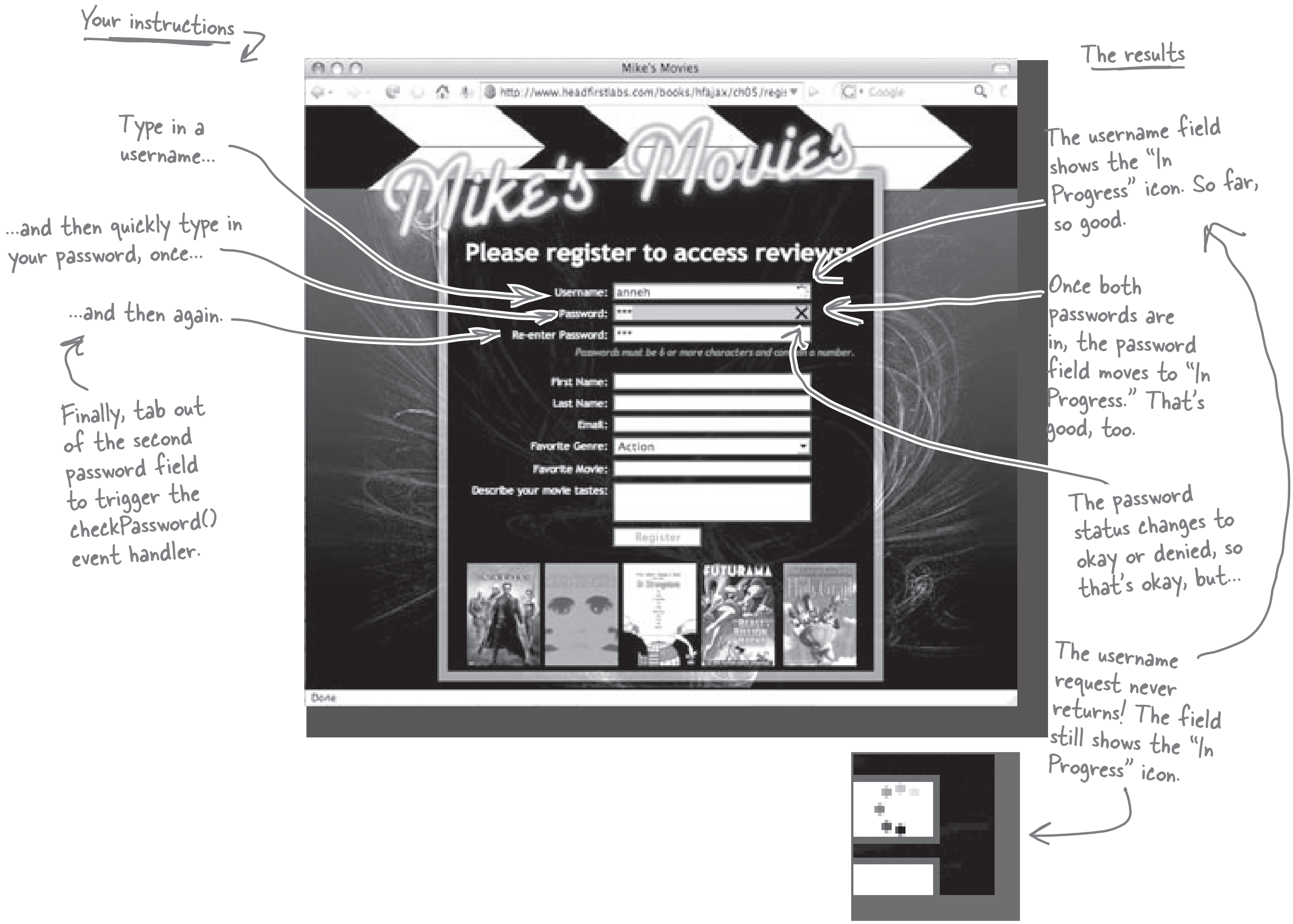
Does anything strange happen? What's going on? What do you think might be causing the problem?



BE the USER Solution

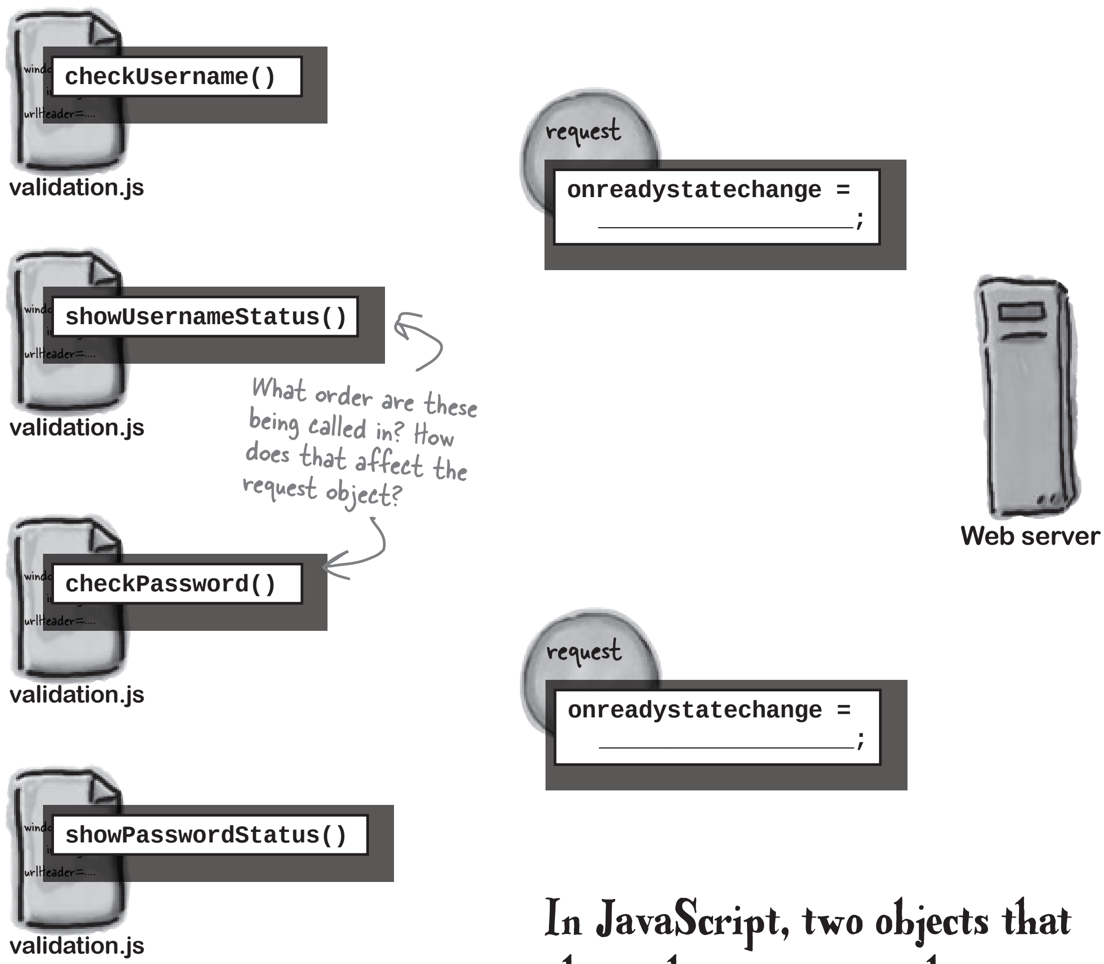
really fast

Your job was to play like you're one of Mike's potential customers... and a fast-typing one, at that. Try and figure out what happens when someone types in a username, and then quickly moves to typing a password in both of the password fields.



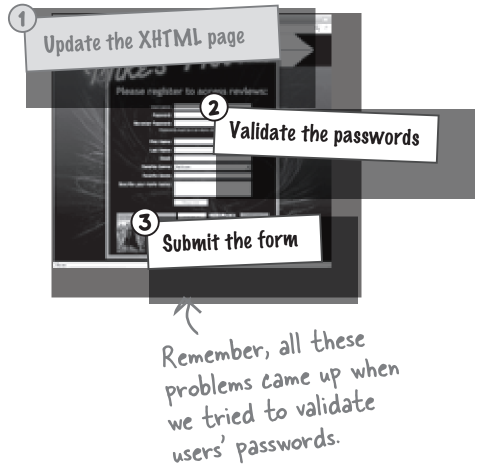
Sharpen your pencil

It's time to figure out what's going on with our asynchronous requests. Below is the request variable named "request", as well as the server. Your job is to draw and label the interactions that are going on between the checkUsername(), showUsernameStatus(), checkPassword(), and showPasswordStatus() functions.

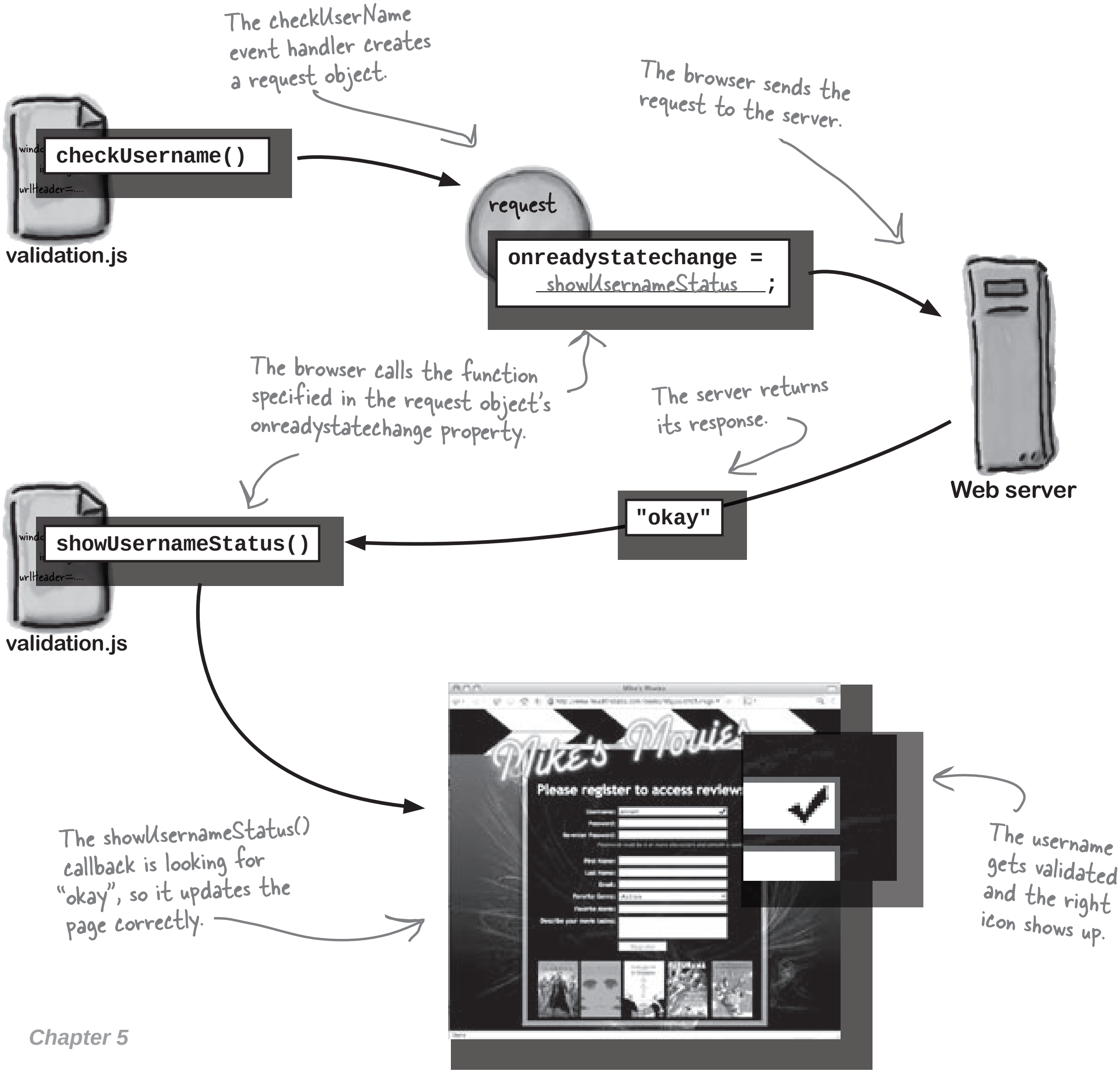


In JavaScript, two objects that share the same name share everything, including property values.

With ONE request object, you can safely send and receive ONE asynchronous request

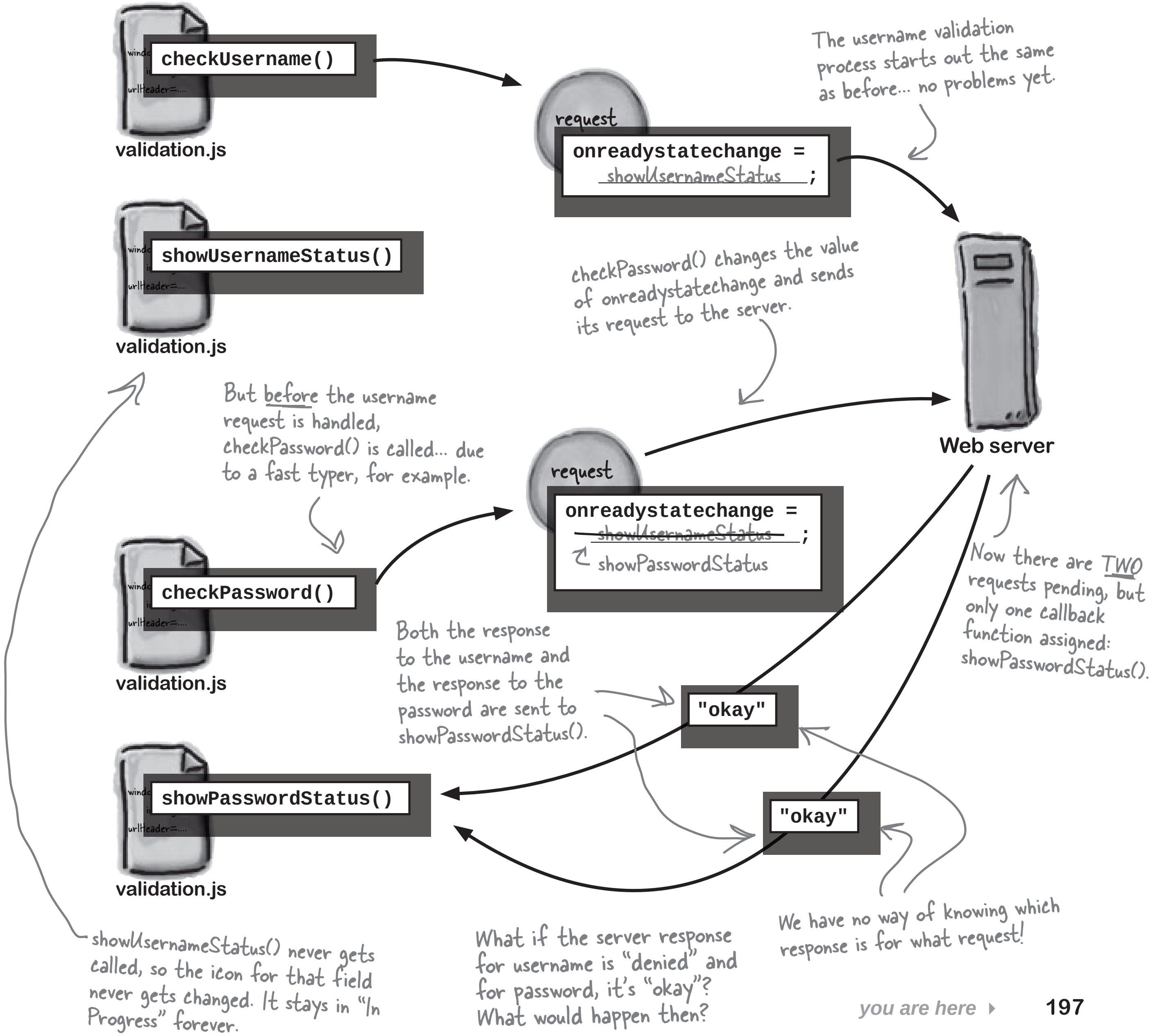


Both `checkUsername()` and `checkPassword()` use the same request object. Because both use the variable name `request`, it's just a single object being used by both. Take a close look at what happens when you're just making a single request, and there's no password validation involved:



Asynchronous requests don't wait on anything... including themselves!

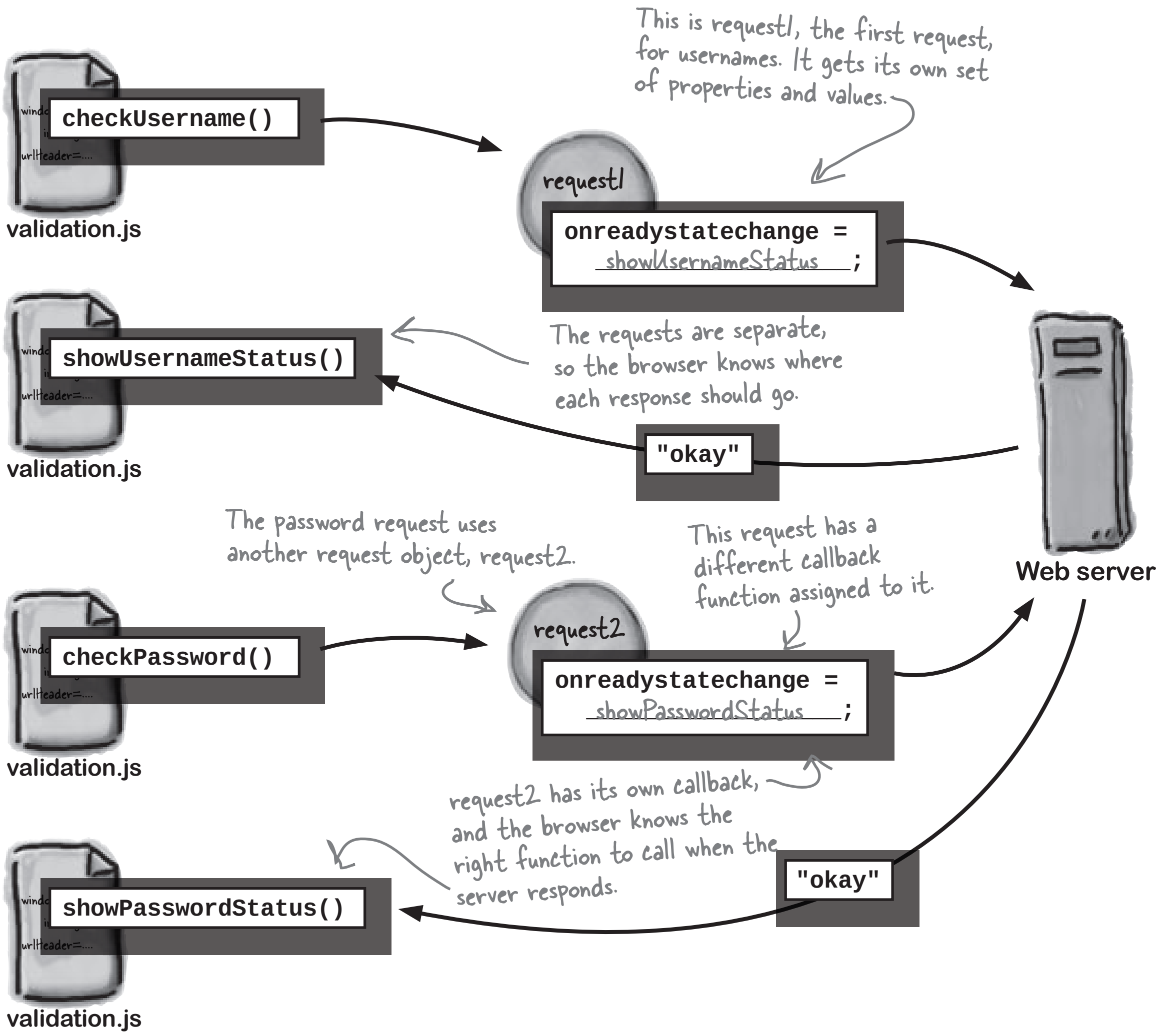
But what happens when there are two requests sharing the same request object? That object can only store one callback function to deal with server responses. That means that you could have two totally different server responses being handled by the *same* callback... and that might not be the *right* callback.



If you're making TWO separate requests, use TWO separate request objects

The problem is that we're using a single request object to make two aynchronous requests. And what does asynchrony mean? That those requests won't wait on a browser or server to get moving. So we end up overwriting one request's data with data from another request.

But what if we have ***two*** asynchronous requests? The two requests won't wait on each other, or make the user wait around, but each request object will have its own data instead of having to share.





You should be ready to update your code to use two request objects. You'll have to change code in validation.js in several different places. See if you can find them all. For username-related requests, use the variable name usernameRequest. For password-related requests, use passwordRequest. When you think you've got them all, turn the page.

there are no Dumb Questions

Q: What does any of this have to do with asynchrony?

A: Well, think about this: what if the request to validate usernames was *not* asynchronous? Then there'd be no way that the password request could get sent before the username request completed. So this problem wouldn't exist in a synchronous environment.

Q: Wouldn't it be easier to just make the username request synchronous?

A: It would be easier, but would that be the best application? Then users would have to wait for their username to get processed. Then, and only then, could they move on to the password field. Sometimes the easiest technical solution is actually the worst usability solution.

Q: Why do the two request variables share property values? Isn't each declared locally within separate functions?

A: It looks that way, but `request` is actually first defined in the `createRequest()` function. Not only that, but `request` is defined in `createRequest()` *without* the `var` keyword. Any variable declared in JavaScript inside a function, but without the `var` keyword, becomes a global variable.

Q: So why not just use the var keyword in createRequest() to fix all of this? Wouldn't that make request local?

A: Good question, but that would cause a different set of problems. If `request` is local, then how would a callback function get access to the request object? The callbacks need `request` to be global, so they can access the variable and its property values.

Q: So how does assigning request to two other variable names help?

A: In JavaScript, assignment is handled by *copying*, and not by *reference*. So when you assign one variable to another value, the new variable gets a copy of the assigned variable. Consider this code:

```
var a = 1;
var b = a;
b = 2;
alert("a = " + a);
alert("b = " + b);
```

You might expect both values to be 2, right? But they're not. When JavaScript interprets `var b = a;`, it creates a new variable named `b`, and puts a copy of `a` into that variable. So no matter what you do to `b`, it won't change `a`.

In the case of the request object, if you create two variables and assign `request` to both, you'll get two *copies* of the original request object. That's two independent request objects that won't affect each other. That's just what we want.

Q: Wow, this is kind of hairy. I'm still confused... what should I do?

A: You may want to pick up a good JavaScript book, like *Head First JavaScript* or *JavaScript: The Definitive Guide*, for more on variable scope and assignment in JavaScript. Or you may want to just follow along, and pick up what you're a little unsure about as you go.

JavaScript considers any variable outside a function, or a variable declared without the var keyword, to be GLOBAL. That variable can be accessed by any function, anywhere.

Exercise Solution

Change all the variable names in checkUserName(), showUsernameStatus(), checkPassword() and showPasswordStatus() fuctions in the registration.js file.

```
function checkUsername() {
  document.getElementById("username").className = "thinking";
  var usernameRequest = createRequest();
  if (usernameRequest == null)
    alert("Unable to create request");
  else {
    var theName = document.getElementById("username").value;
    var username = escape(theName);
    var url= "checkName.php?username=" + username;
    usernameRequest.onreadystatechange = showUsernameStatus;
    usernameRequest.open("GET", url, true);
    usernameRequest.send(null);
  }
}
function showUsernameStatus() {
  if (usernameRequest.readyState == 4) {
    if (usernameRequest.status == 200) {
      if (usernameRequest.responseText == "okay") {
        document.getElementById("username").className = "approved";
        document.getElementById("register").disabled = false;
      } else {
        document.getElementById("username").className = "denied";
        document.getElementById("username").focus();
        document.getElementById("username").select();
        document.getElementById("register").disabled = true;
      }
    }
  }
}
function checkPassword() {
  var password1 = document.getElementById("password1");
  var password2 = document.getElementById("password2");
  password1.className = "thinking";
```

(The "var" before usernameRequest in checkUsername() is struck through.)

It's very important to remove this var... we need usernameRequest to be global, so the callback can reference this variable.

We're using usernameRequest for the request object related to username checks.

Set properties and send the request just like you did before.

Here's why you needed usernameRequest to be global: this callback also has to access the same object.

```
  // First compare the two passwords
  if ((password1.value == "") || (password1.value != password2.value)) {
    password1.className = "denied";
    return;
  }

  // Passwords match, so send request to server
  var passwordRequest = createRequest();
  if (passwordRequest == null) {
    alert("Unable to create request");
  } else {
    var password = escape(password1.value);
    var url = "checkPass.php?password=" + password;
    passwordRequest.onreadystatechange = showPasswordStatus;
    passwordRequest.open("GET", url, true);
    passwordRequest.send(null);
  }
}
function showPasswordStatus() {
  if (passwordRequest.readyState == 4) {
    if (passwordRequest.status == 200) {
      var password1 = document.getElementById("password1");
      if (passwordRequest.responseText == "okay") {
       password1.className = "approved";
       document.getElementById("register").disabled = false;
      } else {
       password1.className = "denied";
       password1.focus();
       password1.select();
       document.getElementById("register").disabled = true;
      }
    }
  }
}
```

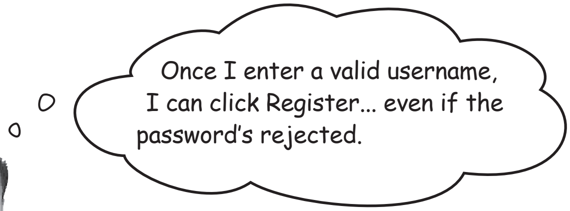
Just like with the other request variable, do not use the var keyword.

passwordRequest is used for all password-related requests.

Now this code has no chance of overwriting properties of the username request object.

BRAIN POWER

There's still a problem with the registration page. Can you figure out what it is?





Validation requires both VERIFICATION and RESTRICTION.

Verification is making sure that a certain piece of data is okay for your system to accept. **Restriction** is not allowing a user to do something until that verification is complete. Good validation combines both of these components.

When we wrote the first version of Mike's page, we disabled the Register button in the `initPage()` function, and re-enabled it once the server validated the user's username. So we *verified* the username and *restricted* the Register button.

But now there's another level of validation: we have to make sure the user's password is okay. Something's going wrong, though... even if a password is rejected, the Register button is getting enabled, and users can click the button.



Validation requires verification AND restriction.

In asynchronous applications, it's not enough to just verify data entered by the user. While that verification is occurring, you have to restrict the user from doing things that depend upon verification.

there are no Dumb Questions

Q: How is enabling the Register button part of restriction? That doesn't make sense...

A: Restriction is the process of not letting a user do something until verification is complete. So part of the restriction process is enabling a button or activating a form. In fact, the end of every restriction process is the *lifting* of that restriction.

Right now, we disable the Register button in initPage()...

The movie page works correctly at the beginning. When the page loads, the Register button is disabled:

```
function initPage() {
  document.getElementById("username").onblur = checkUsername;
  document.getElementById("password2").onblur = checkPassword;
  document.getElementById("register").disabled = true;
}
```

This button is disabled... we (correctly) make sure users can't do anything until they've got a valid username and password.



...and enable the button in the callback functions

We enabled the Register button in the two callback functions, `showUsernameStatus()` and `showPasswordStatus()`. But we're still getting incorrect actions on the form.

showUsernameStatus()

```
if (usernameRequest.responseText == "okay") {
  document.getElementById("username").className = "approved";
  document.getElementById("register").disabled = false;
} else {
  // code to reject username and keep Register disabled
}
```

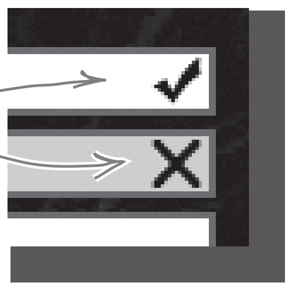
showPasswordStatus()

```
if (passwordRequest.responseText == "okay") {
  password1.className = "approved";
  document.getElementById("register").disabled = false;
} else {
  // code to reject username and keep Register disabled
}
```

Both of these callbacks enable the Register button only when their verification successfully completes.

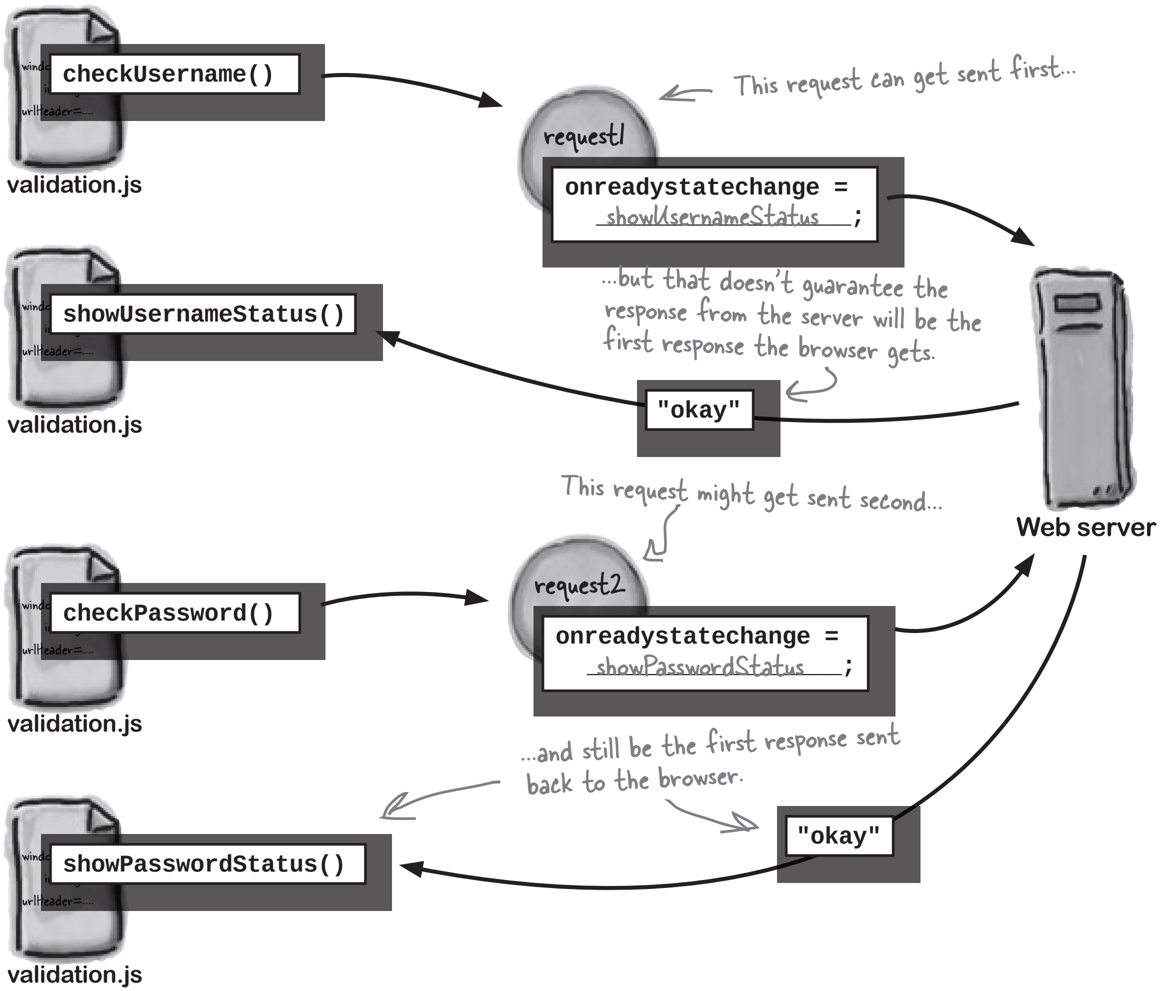


But the Register button is still getting enabled when the username is valid, and the password isn't. What gives?



Asynchrony means you can't count on the ORDERING of your requests and responses

When you send asynchronous requests, you can't be sure of the order that the server will respond to those requests. Suppose that a request to verify a username is sent to the server. Then, another request is sent, this time to verify a password. Which one will return first? ***There's no way of knowing!***



Never count on the ORDER or SEQUENCE of requests and responses in asynchronous applications.

Sharpen your pencil

Can you figure out at least one sequence of requests and responses that would result in the Register button being enabled when either the username or the password is invalid? Draw or list the steps that would have to occur for that to happen.

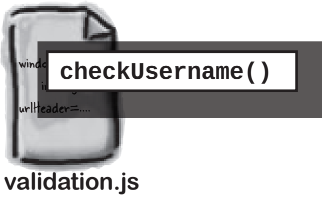
Sharpen your pencil Solution

Here are two different sequences where the Register button ended up enabled when it shouldn't be. Did you come up with one of these? Or something similar?

The Register button always starts out disabled.

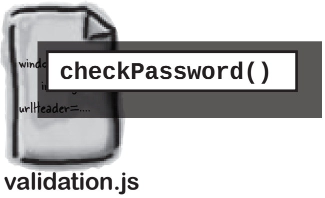


1 The user enters a valid username.





2 The user enters two passwords that don't match.



If the two passwords don't match, there's no request to the server... so there's an almost instant denied result.

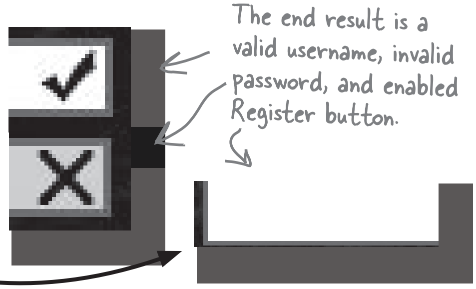
3 The password field shows the "denied" X mark.



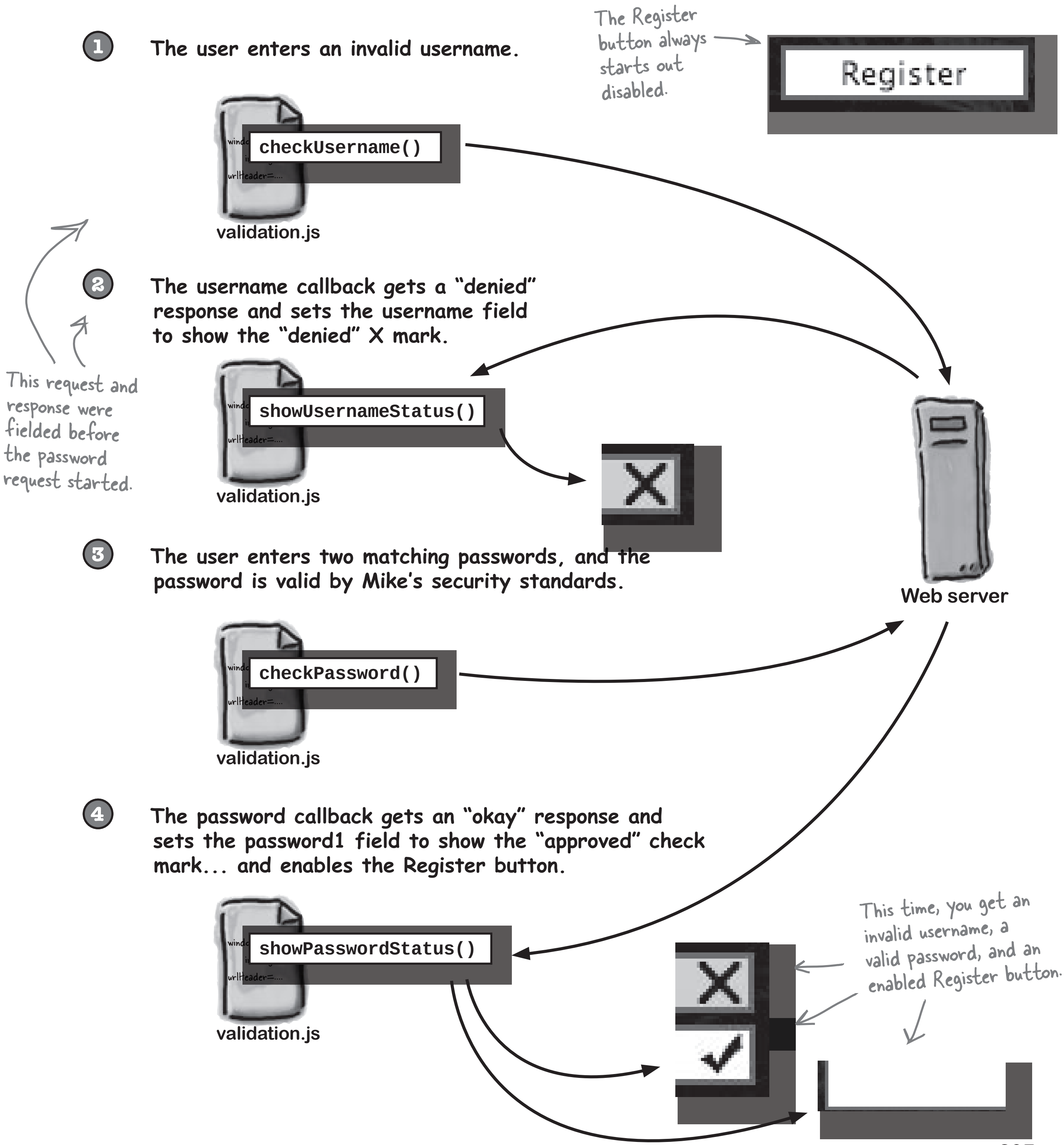
The server returns its response after the passwords have already been denied.

4 The username callback gets an "okay" response and sets the username field to show the "approved" check mark... and enables the Register button.





The end result is a valid username, invalid password, and enabled Register button.



1 The user enters an invalid username.
The Register button always starts out disabled.
Register
checkUsername()
validation.js
2 The username callback gets a "denied" response and sets the username field to show the "denied" X mark.
This request and response were fielded before the password request started.
showUsernameStatus()
validation.js
Web server
3 The user enters two matching passwords, and the password is valid by Mike's security standards.
checkPassword()
validation.js
4 The password callback gets an "okay" response and sets the password1 field to show the "approved" check mark... and enables the Register button.
showPasswordStatus()
validation.js
This time, you get an invalid username, a valid password, and an enabled Register button.



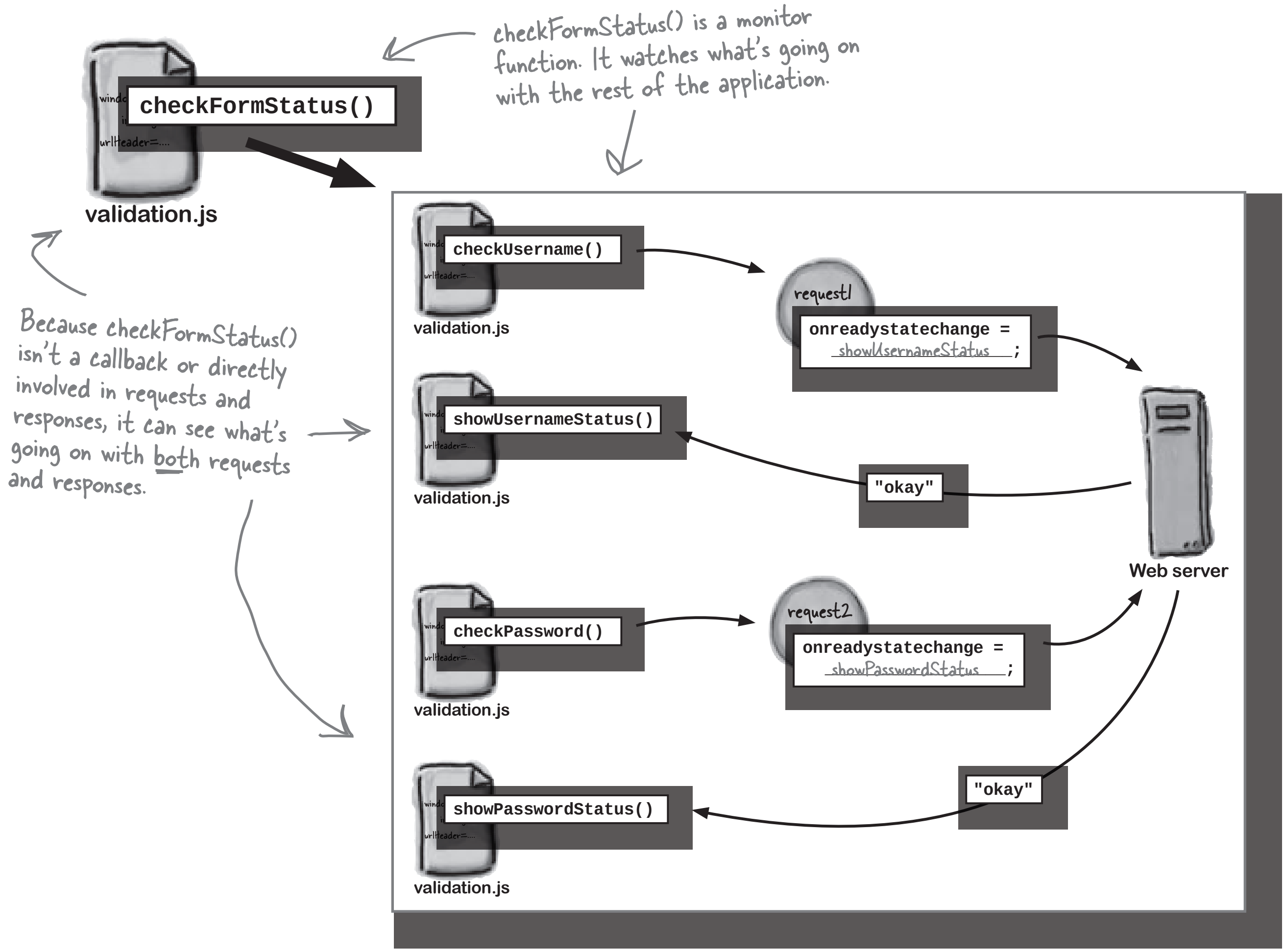
Good usability is a pain to create.

No matter how you cut it, building an application that's highly usable is hard work. In this case, we added asynchrony to make Mike's registration page more usable. Users can keep typing in their information while the server's validating their username and password.

But now all that asynchrony is creating some problems. What we need is a way to know when both the username *and* password are valid. Then—and ***only*** then—we can enable the Register button. We need a way to monitor the status of each field and make sure something happens only when both fields are approved.

A monitor function MONITORS your application... from OUTSIDE the action

We need a monitor function. That's a function that monitors certain variables or parts of an application, and then takes action based on the things it's monitoring.



Exercise

Can you figure out what a checkFormStatus() monitor function should do? You'll also need to call that function. Where in your code should that happen? If you're not sure, think about it for a while... and then turn the page for a few helpful hints.

You call a monitor function when action MIGHT need to be taken

Monitor functions are usually used to update a part of an application or page that depends on several variables. So you call the monitor when you think it ***might*** be time to update a page... like when a username or password comes back approved.

Right now, the username and password callbacks directly update the Register button's status

The problem we're having now is that in `showUsernameStatus()` and `showPasswordStatus()`, we're updating the Register button. But neither of those functions really have ***all*** the information they need to update that button.

showUsernameStatus()

```
if (usernameRequest.responseText == "okay") {
  document.getElementById("username").className = "approved";
  document.getElementById("register").disabled = false;
} else {
  // code to reject username and keep Register disabled
}
```

showPasswordStatus()

```
if (passwordRequest.responseText == "okay") {
  password1.className = "approved";
  document.getElementById("register").disabled = false;
} else {
  // code to reject password and keep Register disabled
}
```



This enabling is being done with incomplete information! The username callback doesn't check to see if the password is valid, and the password callback doesn't check to see if the username is valid. The result: the button gets enabled when it shouldn't be.

Let's have the callbacks run the monitor function...

So instead of directly changing the button status, we can change our callback functions to run the monitor function. That way, it's not up to either callback to figure out what status the Register button should be in.

showUsernameStatus()

```
if (usernameRequest.responseText == "okay") {
  document.getElementById("username").className = "approved";
  document.getElementById("register").disabled = false;
  checkFormStatus();
} else {
  // code to reject username and keep Register disabled
}
```

Remove the lines in both callbacks that updated the status of the Register button.

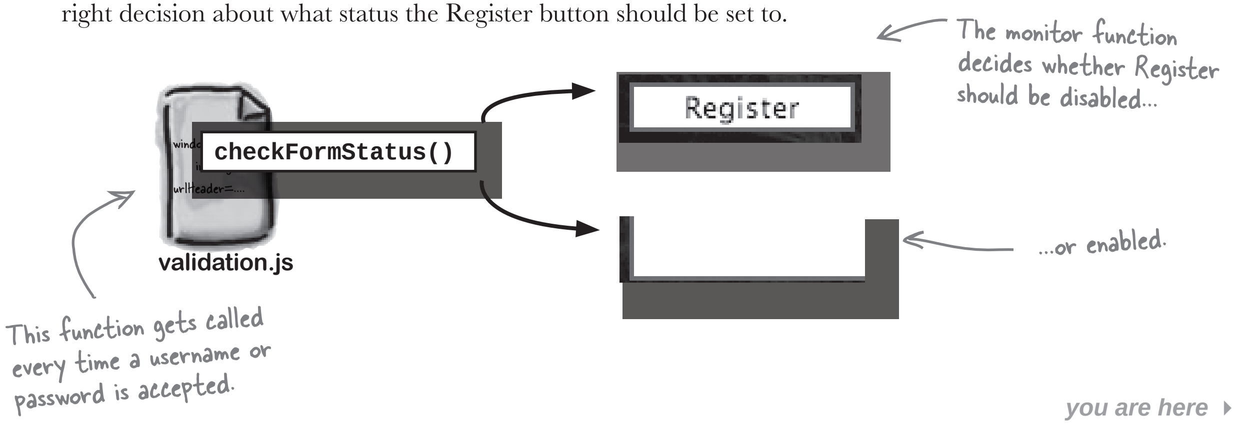
showPasswordStatus()

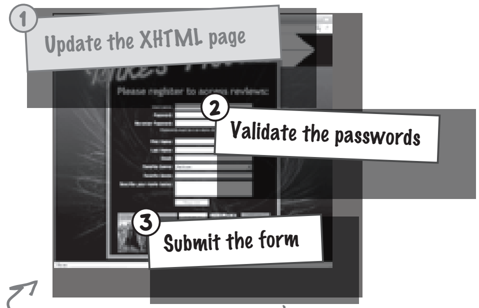
```
if (passwordRequest.responseText == "okay") {
  password1.className = "approved";
  document.getElementById("register").disabled = false;
  checkFormStatus();
} else {
  // code to reject username and keep Register disabled
}
```

Now both callbacks should call the monitor function, checkFormStatus().

...and let the monitor function update the Register button

Since the monitor function is separate from either the username or password checks, it can get all the information it needs. The monitor function can check the username and password fields, and make the right decision about what status the Register button should be set to.





Believe it or not, we're still working on getting the password functionality right.

Status variables let monitors know what's going on

We're ready to write a monitor function to set the status of the Register button's `disabled` property, and now both callbacks call that monitor. All that's left is to have those callbacks set some status variables, indicating whether the username and password are valid. The monitor function can use those variables to figure out what to do when it's called.

Here's the complete code for Mike's app, with a new monitor function:

```
window.onload = initPage;
var usernameValid = false;
var passwordValid = false;

function initPage() { // initPage stays the same }
function checkUsername() { // checkUsername stays the same }

function showUsernameStatus() {
  if (usernameRequest.readyState == 4) {
    if (usernameRequest.status == 200) {
      if (usernameRequest.responseText == "okay") {
        document.getElementById("username").className = "approved";
        document.getElementById("register").disabled = false;
        usernameValid = true;
      } else {
        document.getElementById("username").className = "denied";
        document.getElementById("username").focus();
        document.getElementById("username").select();
        document.getElementById("register").disabled = true;
        usernameValid = false;
      }
      checkFormStatus();
    }
  }
}

function checkPassword() {
  var password1 = document.getElementById("password1");
  var password2 = document.getElementById("password2");
  password1.className = "thinking";

  // First compare the two passwords
  if ((password1.value == "") || (password1.value != password2.value)) {
    password1.className = "denied";
    passwordValid = false;
```

We need two new global variables. usernameValid is the current status of the username, and passwordValid is the current status of the password.

We're using var, but we're declaring these outside of any function. That means they're global variables.

We need to update usernameValid for both possible server responses.

We don't want to change the status of the Register button in either of the if/else branches.

Since we need to call the monitor function in either case, it's easier to leave it outside the if/else statement.

This is easy to forget about, but if the passwords don't match, we need to update the passwordValid status variable.



validation.js

```
    checkFormStatus();
    return;
  }

  // Passwords match, so send request to server
  passwordRequest = createRequest();
  if (passwordRequest == null) {
    alert("Unable to create request");
  } else {
    var password = escape(password1.value);
    var url = "checkPass.php?password=" + password;
    passwordRequest.onreadystatechange = showPasswordStatus;
    passwordRequest.open("GET", url, true);
    passwordRequest.send(null);
  }
}

function showPasswordStatus() {
  if (passwordRequest.readyState == 4) {
    if (passwordRequest.status == 200) {
      var password1 = document.getElementById("password1");
      if (passwordRequest.responseText == "okay") {
        password1.className = "approved";
        document.getElementById("register").disabled = false;
        passwordValid = true;
      } else {
        password1.className = "denied";
        password1.focus();
        password1.select();
        document.getElementById("register").disabled = true;
        passwordValid = false;
      }
      checkFormStatus();
    }
  }
}

function checkFormStatus() {
  if (usernameValid && passwordValid) {
    document.getElementById("register").disabled = false;
  } else {
    document.getElementById("register").disabled = true;
  }
}
```

None of our code should set the status of the Register button except for the monitor.

This is just like the username callback. Update the global status variable for password...

...and then call the monitor function.

All this function has to do is check the two status variables...

...and set the Register button's status accordingly.

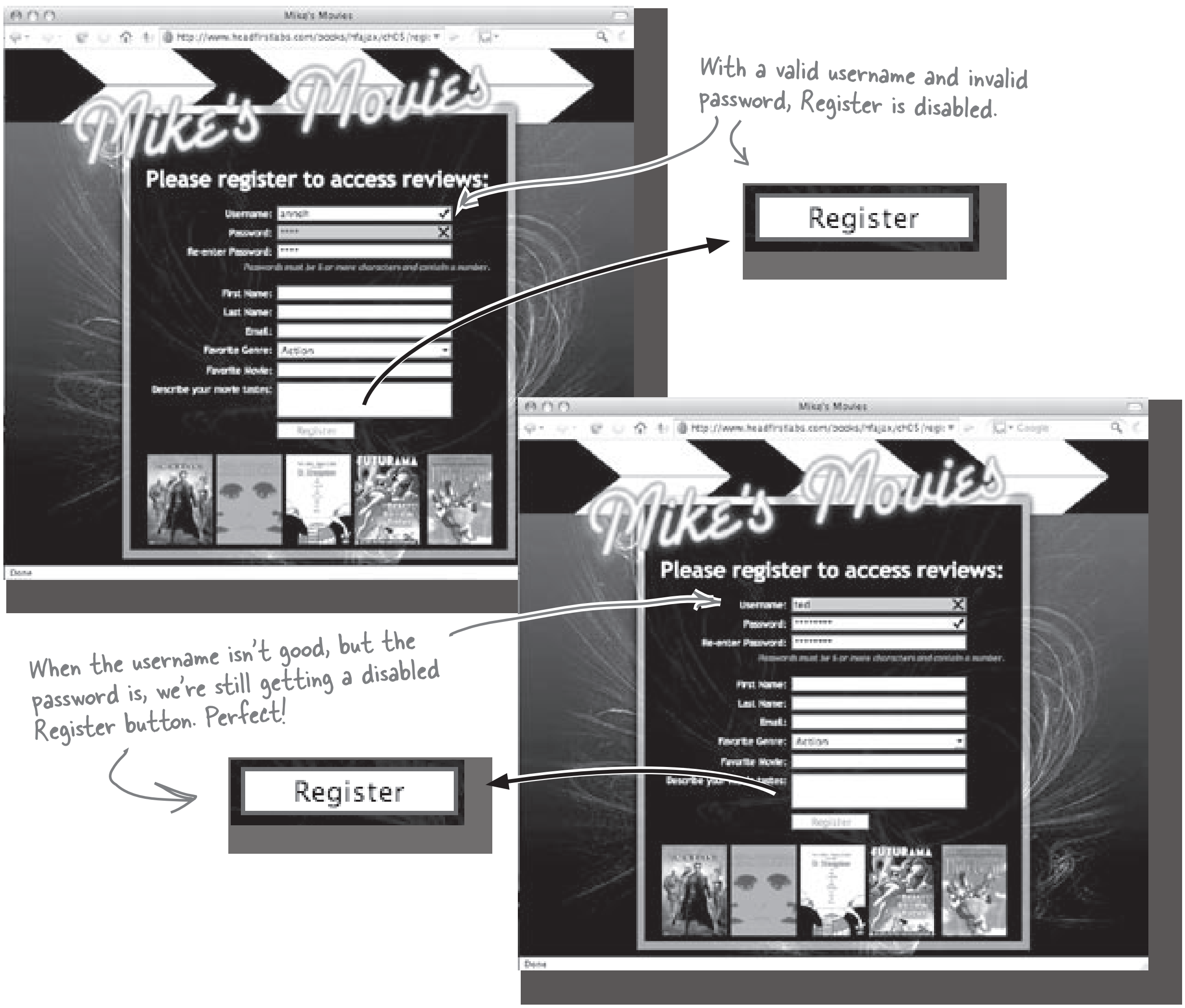
Explicitly set the button to disabled, in case there was a valid username or password before, but now there's a change that makes one of those invalid.

Test Drive

Finally! But does it all work?

Make sure your version of validation.js matches the version shown on the last two pages. You should have two new global variables, an updated version of `checkPassword()`, two updated callback functions, and a new monitor function, `checkFormStatus()`.

Load everything up. Try out the scenarios you worked out for the exercise from page 205. Do they still break the page? If not, you've solved Mike's asynchrony problems!



there are no Dumb Questions

Q: Can you explain what a monitor function is again?

A: Sure. A monitor function is just a function that monitors your application. So for Mike's registration page, the monitor function is monitoring the state of the username and password variables, and it's changing the form to match the current status.

Q: I thought monitor functions usually ran automatically, like at set intervals.

A: Sometimes they do. In systems where you have a lot more threading capability—the ability to run a process in the background—it's common to have a monitor function execute periodically. Then, you don't have to explicitly call the monitor, which is what we do in the username and password callbacks.

Refactoring code is pulling out common parts and putting those parts into a single, easily-maintainable function or method. Refactoring makes code easier to update and maintain.

Q: Why didn't you declare usernameValid and passwordValid in initPage()?

A: You could do that. But if you do declare the variables inside `initPage()`, be sure *not* to use the `var` keyword. `usernameValid` and `passwordValid` need to be global variables.

Variables declared *outside* of any function (with or without `var`) are global. Variables declared *inside* a function, but *without* `var`, are also global. And variables declared *inside* a function, *with* `var`, are local. It's a bit confusing, that's for sure.

In fact, that's why they're left outside of any function: it makes it a little clearer that those two variables are global, and not local to any particular function.

Q: So then why aren't usernameRequest and passwordRequest declared there also?

A: That's actually a good idea, and you might want to make that change. In our code, we left them in `checkUsername()` and `checkPassword()` because that's where those variables were originally created (back when they were both called `request`).

Q: Couldn't I set the status of the username and password1 fields in my monitor function, too?

A: You sure could. In fact, that's probably a good idea. That would mean that there'd be less CSS class-changing happening all over the code. Most of that display logic would be handled by the monitor, which is already dealing with the display of the Register button.

Anytime you can consolidate (or **refactor**) code without a lot of ill consequences, it's a good idea. Cleaner code is easier to modify and maintain.

Q: Just adding in a password field sure made things complicated. Is that normal?

A: In asynchronous apps, adding an additional asynchronous request is usually pretty tricky. The thing that added a lot of complexity to Mike's app wasn't the additional password *fields*, but the additional *request* we needed to make to deal with those fields.

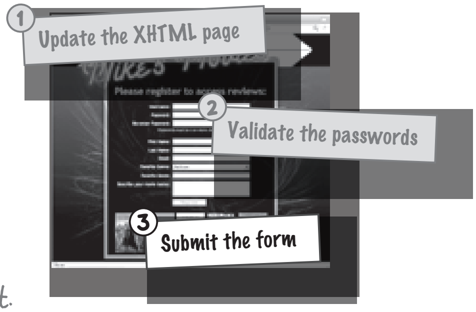
Q: And this is all just so users can keep typing instead of waiting?

A: It sure is. You'd be surprised at how impatient web users are (or maybe you wouldn't!). Typing in a username, waiting for the username to get validated, typing in a password, and then also waiting for the password to get validated... that's a lot of waiting. It's even worse that after all that waiting, the user still has to fill out the rest of the form.

Saving a couple of seconds here and there really adds up on the Web. In fact, it might be the difference between keeping a customer and losing them.

Q: So what about form submits? There's going to be waiting there, too, right?

A: Now you're getting ahead! But that's exactly what Mike was thinking when he asked for scrolling images...



And now for our last trick...

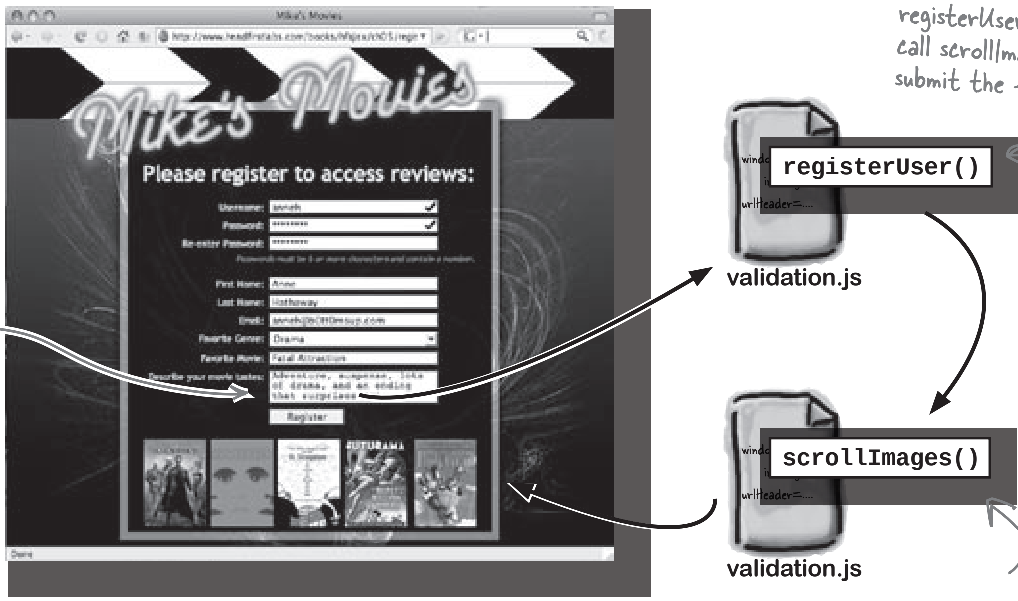
Mike's got one last request. When users click the Register button, the images along the bottom should begin to scroll while the form is processing. This gives the user something interesting to watch while they're waiting on Mike's registration logic.

And now we know the Register button works right.

Fortunately, this shouldn't be too difficult. Here's what we need to do to put this into action:

Let's create a new function, registerUser(), to call scrollImages() and submit the form.

Instead of letting the form submit via a "submit" button, let's assign a click handler to the Register button.



We can abstract the code for animating the images into another function, scrollImages(), and call that when we need to scroll the images.

Sharpen your pencil

Do you think the request to submit the form to Mike's server should be synchronous or asynchronous?

☐ Synchronous ☐ Asynchronous

Why? ....

....

....

Does your choice above have any affect on the scrolling of the images along the bottom of the page?

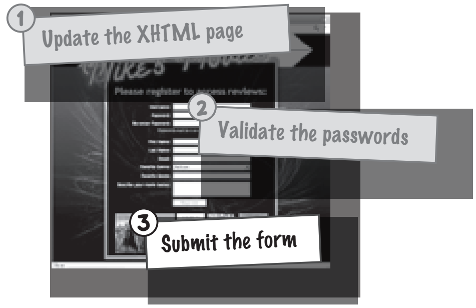
....

....

....

....

The answers to these questions are spread out over the rest of the chapter, so you'll have to keep reading to find out if you got these right.



Synchronous requests block ALL YOUR CODE from doing anything

When you send an entire form off to be processed, you usually want that request to be **synchronous**. That's because you don't want users to change that data while the server is working with it.

But Mike wants scrolling images *while* the user is waiting on the server. That means you need your code to run *while the server is working on a response*. So even though the request would ideally be synchronous, you need it to be an ***asynchronous*** request to fulfill image-loving Mike's needs.

This isn't a perfect solution, but lots of times you've got to make this sort of choice: satisfying the client's needs even when the result is a little less than ideal. Mike's willing to let users mess around with the form, if they really want to, while their request is being processed. He figures they won't do that, though, ***because*** of the scrolling images. They'll be too busy thinking about which movie review they want to check out when they're logged in.

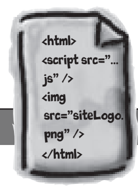
First, we no longer need a "submit" button

A "submit" button in XHTML submits a form. And since we no longer need the Register button to submit the form, we can make it a normal button. Then, we can submit the form in our JavaScript.

```
      <li><label for="favorite">Favorite Movie:</label><input id="favorite"
                 type="text" name="favorite" /></li>
      <li><label for="tastes">Describe your movie tastes:</label><textarea
                 name="tastes" cols="60" rows="2" id="tastes"></textarea></li>
      <li><label for="register"></label><input id="register"
                 type="button" value="Register" name="register" /></li>
    </ul>
  </form>

  <!-- Cover images -->
</div>
</body>
</html>
```

We need a regular button now, not a submit button.



registration.html

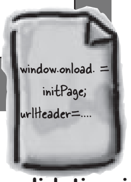
The original version of our XHTML was shown way back on page 182.

Second, we need to register a new event handler for the button's onclick event

Now we need to attach an event handler to that button. We'll call the function we want to run `registerUser()`, and we can make the assignment in `initPage()`:

```
function initPage() {
  document.getElementById("username").onblur = checkUsername;
  document.getElementById("password2").onblur = checkPassword;
  document.getElementById("register").disabled = true;
  document.getElementById("register").onclick = registerUser;
}
```

Assign the new callback we'll write to the Register button's onclick event.



validation.js

Third, we need to send an ASYNCHRONOUS request to the server

Finally, we need a new event handler function. This function needs to get a new request object, and send it to the server. And this should be an asynchronous request, so we can animate and scroll those images while the user is waiting.

Let's change the text on the button to provide a little information to the user.

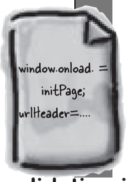
```
function registerUser() {
  document.getElementById("register").value = "Processing...";
  registerRequest = createRequest();
  if (registerRequest == null) {
    alert("Unable to create request.");
  } else {
    var url = "register.php?username=" +
      escape(document.getElementById("username").value) + "&password=" +
      escape(document.getElementById("password1").value) + "&firstname=" +
      escape(document.getElementById("firstname").value) + "&lastname=" +
      escape(document.getElementById("lastname").value) + "&email=" +
      escape(document.getElementById("email").value) + "&genre=" +
      escape(document.getElementById("genre").value) + "&favorite=" +
      escape(document.getElementById("favorite").value) + "&tastes=" +
      escape(document.getElementById("tastes").value);
    registerRequest.onreadystatechange = registrationProcessed;
    registerRequest.open("GET", url, true);
    registerRequest.send(null);
  }
}
```

Create another request object...

...and configure the object's properties.

This is all new code.

It's tempting to make this synchronous, but that would block the image scrolling we're going to add in a few pages.



validation.js



Usability is in the eye of the ~~beholder~~... err... the client.

Sometimes clients do things that don't make sense to you. That's why they're paying the bills, and you're collecting the checks. You can suggest alternatives to your client, but at the end of the day, you're going to be a lot happier if you just build the client what they ask for.

In Mike's case, he wants to entice users with reviews available on his site, so he wants images to scroll while users are waiting on their registration request. That makes his form a little less usable, though. Now, instead of waiting on a response, users can actually type over their entries. That could create some confusion about what information Mike's system actually registered for that user.

Then again, Mike will probably just call you later when he realizes that for himself... and that's not altogether a bad thing, is it?

You can suggest alternative ideas to your clients, but ultimately, you should almost <u>ALWAYS</u> build what the client asked for... even if you don't agree with their decisions.

there are no Dumb Questions

Q: Could I disable all the fields while the images are scrolling?

A: That's a great idea! Why don't you take some time now to do that. Mike will love that he gets scrolling, and you'll still keep the nice usability you've built into the registration page so far. We won't show that code, though, so consider it a little extra-credit project.

Use setInterval() to let JavaScript run your process, instead of your own code

`setInterval()` is a handy method that lets you pass in a function, and have the JavaScript interpreter run your code over and over, every so often. Since it's the interpreter running your code, the function you send `setInterval()` will run even while your code is busy doing other things like, say, registering a user.

To use `setInterval()`, you pass it a function to execute and the interval at which the function should be called, in milliseconds. The method returns a **token**, which you can use to modify or cancel the process.

1 second is 1000 milliseconds.

Here's `setInterval()` in action.

```
t = setInterval(scrollImages, 50);
```

This is the token that you can use to cancel the interval.

setInterval() is the method itself.

The first argument to setInterval() is the statement to be evaluated. In this case, we want it to call the function called scrollImages. You leave off the parentheses so that JavaScript will actually reference the function, not just run it once.

This tells JavaScript how often to execute the statement. We've chosen 40 milliseconds, which is a good average rate to scroll something.

You can use any valid JavaScript here, including an anonymous function.

there are no Dumb Questions

Q: Is the function you pass to setInterval() a callback?

A: Yes. Every time the interval you set passes, the function you pass in here will be called back by the browser.

Q: So do you write that function just like the callback for a request object?

A: Well, there isn't a request object involved, and so you don't need to check any `readyState` or `status` properties. And there's no server response to evaluate. So you just need a JavaScript function that does something every time it's called.

Q: So I can do anything inside a setInterval() callback that I can do in JavaScript?

A: Yes, that's right. There's no limitation on what you can do inside the function.

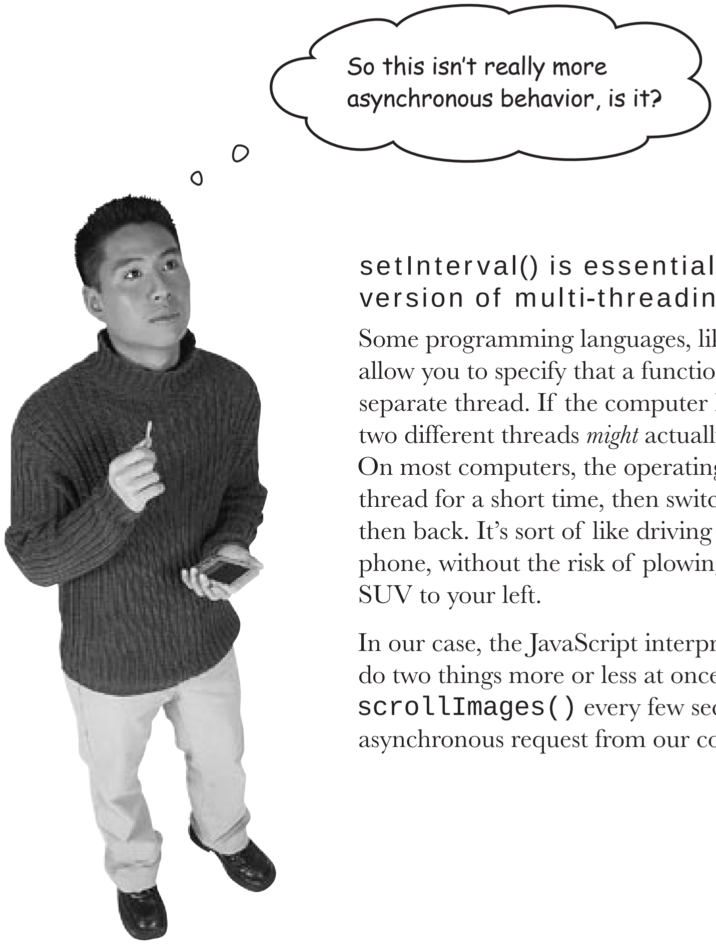
Q: Why do you use the parentheses when you specify the function?

A: Because you're not setting a property, like you do when you assign an event handler. You're actually passing code to the JavaScript interpreter. The interpreter will then execute that code every time the interval elapses.

Q: How many times does the callback happen?

A: Until you cancel the timer. You can do that with `clearInterval()`.

You can pass any JavaScript function to setInterval(), and have it run automatically at pre-determined intervals.



setInterval() is essentially JavaScript's version of multi-threading.

Some programming languages, like C# and JavaScript, allow you to specify that a function be executed on a separate thread. If the computer has more than one CPU, two different threads *might* actually execute simultaneously. On most computers, the operating system executes one thread for a short time, then switches to another thread, then back. It's sort of like driving and talking on a cell phone, without the risk of plowing into the guy in the big SUV to your left.

In our case, the JavaScript interpreter is able to do two things more or less at once: keep executing `scrollImages()` every few seconds, and deal with the asynchronous request from our code to Mike's web server.



You're in the home stretch now. There are just a few things left to do to finish up Mike's registration page... and you're going to do them all right now. Here's your list of things to take care of before turning the page:

- [ ] Add the following Ready-Bake Code for scrolling the cover images to `validation.js`.

```
function scrollImages() {
  var coverBarDiv = document.getElementById("coverBar");
  var images = coverBarDiv.getElementsByTagName("img");
  for (var i = 0; i < images.length; i++) {
    var left = images[i].style.left.substr(0,
               images[i].style.left.length - 2);
    if (left <=  -86) {
      left = 532;
    }
    images[i].style.left = (left - 1) + "px";
  }
}
```

Find all the images.

For each image, figure out what its current position is using the "left" attribute of its style property...

...and then move the image just a bit further to the left (or loop it around).



We don't explain this code because it's standard JavaScript. You can use it safely, though... and dig into Head First JavaScript for more details.

- [ ] Add a line to the Register button's event handler callback that tells the JavaScript interpreter to run `scrollImages()` every 50 milliseconds.

- [ ] Write a callback function for the asynchronous registration request. When the callback gets a response from the server, it should replace the "wrapper" `<div>`'s content with the server's response. You can assume the server returns an XHTML fragment suitable for display.

- [ ] Test your code out before turning the page. You can do this!

Make sure you've got the CSS from the Chapter 5 examples. The earlier version of Mike's CSS doesn't have styles for the cover images.



Your job was to complete Mike's registration page. Did you figure everything out? Here's how we finished up the page.

```
function registerUser() {
  t = setInterval("scrollImages()", 50);
  document.getElementById("register").value = "Processing...";
  registerRequest = createRequest();
  if (registerRequest == null) {
    alert("Unable to create request.");
  } else {
    var url = "register.php";
    registerRequest.onreadystatechange = registrationProcessed;
    registerRequest.open("GET", url, true);
    registerRequest.send(null);
  }
}

function registrationProcessed() {
  if (registerRequest.readyState == 4) {
    if (registerRequest.status == 200) {
      document.getElementById('wrapper').innerHTML =
        registerRequest.responseText;
    }
  }
}

function scrollImages() {
  var coverBarDiv = document.getElementById("coverBar");
  var images = coverBarDiv.getElementsByTagName('img');
  for (var i = 0; i < images.length; i++) {
    var left = images[i].style.left.substr(0, images[i].style.left.length - 2);
    if (left <=  -86) {
      left = 532;
    }
    images[i].style.left = (left - 1) + 'px';
  }
}
```

Here's where we start the animation.

This callback is pretty simple. It gets a response and replaces the content of main <div> on the page with that response.

This is the Ready Bake Code from the last page. It handles scrolling the images.

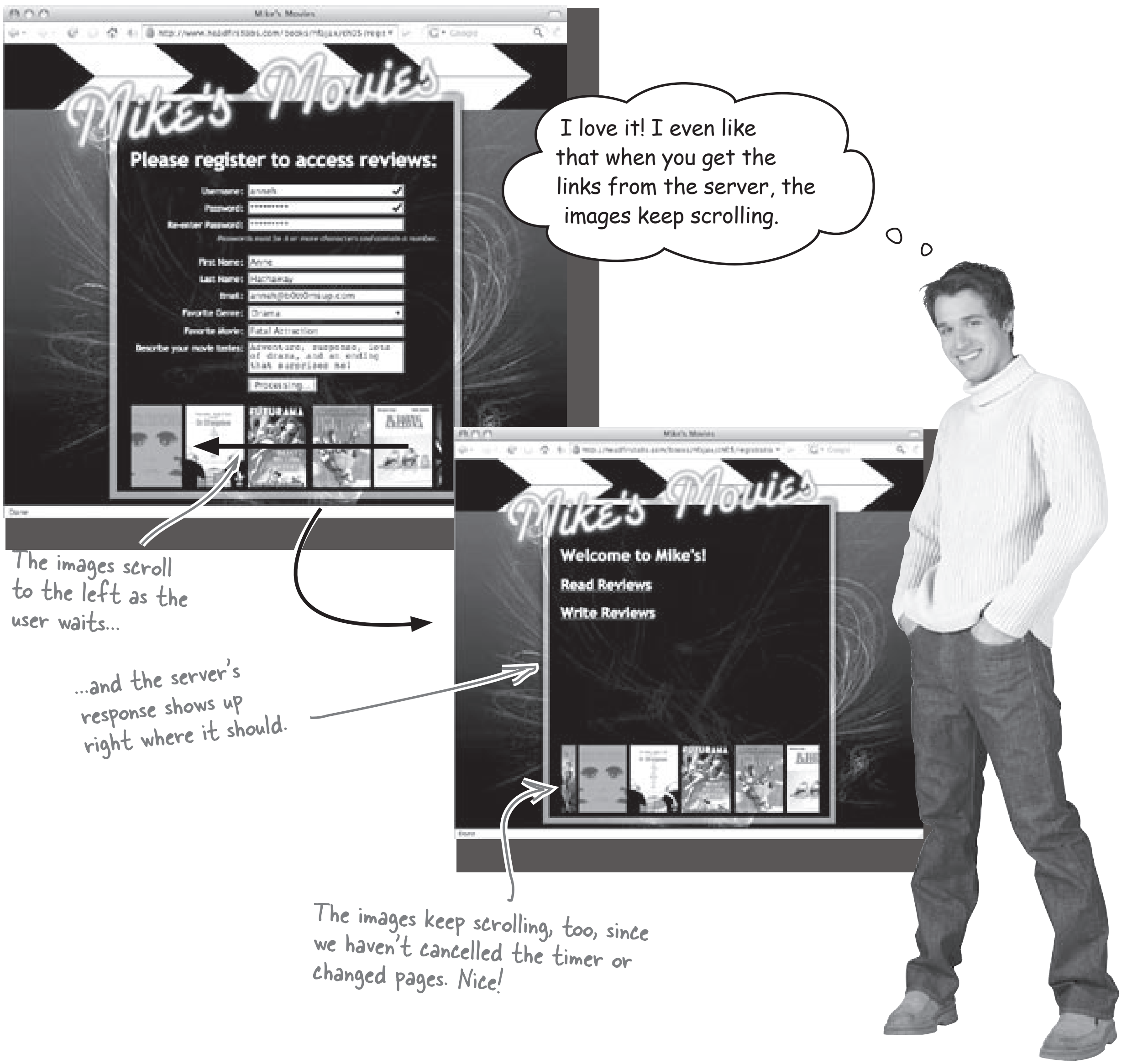


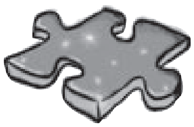
validation.js

Test Drive

Let's show Mike what we've done.

It's been a long, fast ride since all Mike cared about was validating usernames. We've added a lot... let's show him what we've come up with.





Word Search

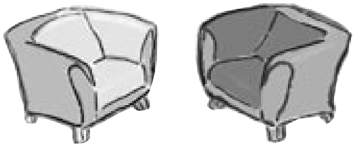
Take some time to sit back and give your right brain something to do. See if you can find the key words below in the letter scramble. Good luck!

X	P	R	S	M	O	K	E	J	U	D	H	E
A	A	L	A	V	R	E	T	N	I	T	E	S
A	S	I	O	R	E	M	A	L	T	R	T	V
Q	S	L	X	H	A	N	D	L	E	R	S	L
C	W	Y	O	R	U	H	A	E	A	Y	E	R
A	O	N	N	O	N	T	L	B	N	E	U	A
L	R	E	U	C	S	T	F	C	L	N	Q	S
L	D	T	K	A	H	P	H	N	L	E	E	N
B	E	Y	C	C	E	R	L	R	X	L	R	R
A	N	I	T	H	O	E	O	A	E	D	G	R
C	U	N	B	N	D	Q	B	N	R	A	A	K
K	N	G	O	F	E	U	R	L	O	N	D	A
N	D	U	L	R	I	V	F	R	I	U	D	Y
A	S	E	A	D	I	V	E	A	T	E	S	D
J	E	R	C	I	C	T	H	R	I	Z	A	R

Word list:

setInterval
Asynchronous
Synchronous
DIV
Handlers
Callback
Thread
Password
Event
Request
Enable

Fireside Chats



Tonight's talk: **Asynchronous and Synchronous applications go toe-to-toe**

Synchronous: Hey there, long time, no talk.

Asynchronous: No kidding. Every time I call you, I get a busy signal.

Synchronous: I'm a busy guy, you know? And I don't let anything get in the way of paying attention to the user I'm serving.

Asynchronous: But what about all your *other* users? They're just left waiting around?

Synchronous: They'll get their turn, too. Sometimes it's much better to take care of one thing at a time, and then move on to the next job. Slow and steady...

Asynchronous: You can say ***that*** again!

Synchronous: Just because I don't let people interrupt me while I'm working—

Asynchronous: Hey, I can listen ***and*** talk, all at the same time. You're the one with the one-track mind.

Synchronous: One-track mind? I just make sure I finish what I start.

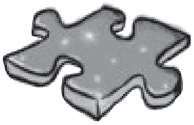
Asynchronous: Sure, but what if that takes 10 seconds? Or 10 minutes? Or an hour? Do you really think people enjoy that little hourglass whirling around?

Synchronous: I don't seem to get too many complaints.

Asynchronous: Yeah, well, I'd love to sit around like this all day, but my users don't like to wait on me. That's more your department, isn't it?

Synchronous: Hey, enjoy your 15 minutes of fame, bro. I've seen fads like you come and go a million times.

Asynchronous: I bet you thought U2 was a one-hit wonder, too. I'm not going anywhere—except to make the Web a hip place again. See you when I see you...



Word Search Solution

X	P	R	S	M	O	K	E	J	U	D	H	E
A	A	L	A	V	R	E	T	N	I	T	E	S
A	S	I	O	R	E	M	A	L	T	R	T	V
Q	S	L	X	H	A	N	D	L	E	R	S	L
C	W	Y	O	R	U	H	A	E	A	Y	E	R
A	O	N	N	O	N	T	L	B	N	E	U	A
L	R	E	U	C	S	T	F	C	L	N	Q	S
L	D	T	K	A	H	P	H	N	L	E	E	N
B	E	Y	C	C	E	R	L	R	X	L	R	R
A	N	I	T	H	O	E	O	A	E	D	G	R
C	U	N	B	N	D	Q	B	N	R	A	A	K
K	N	G	O	F	E	U	R	L	O	N	D	A
N	D	U	L	R	I	V	F	R	I	U	D	Y
A	S	E	A	D	I	V	E	A	T	E	S	D
J	E	R	C	I	C	T	H	R	I	Z	A	R

Word list:

setInterval
Asynchronous
Synchronous
DIV
Handlers
Callback
Thread
Password
Event
Request
Enable

6 the document object model

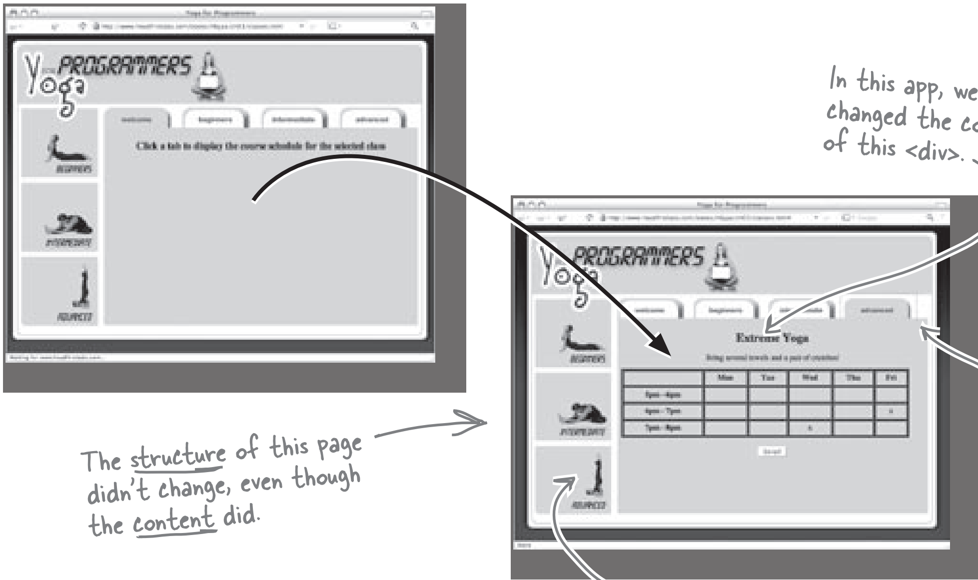
Web Page Forestry



Wanted: easy-to-update web pages. It's time to take things into your own hands and start writing code that updates your web pages on the fly. Using the **Document Object Model**, your pages can take on new life, responding to users' actions, and you can ditch unnecessary page reloads forever. By the time you've finished this chapter, you'll be able to find, move, and update content virtually anywhere on your web page. So turn the page, and let's take a stroll through the Webville Tree Farm.

You can change the CONTENT of a page...

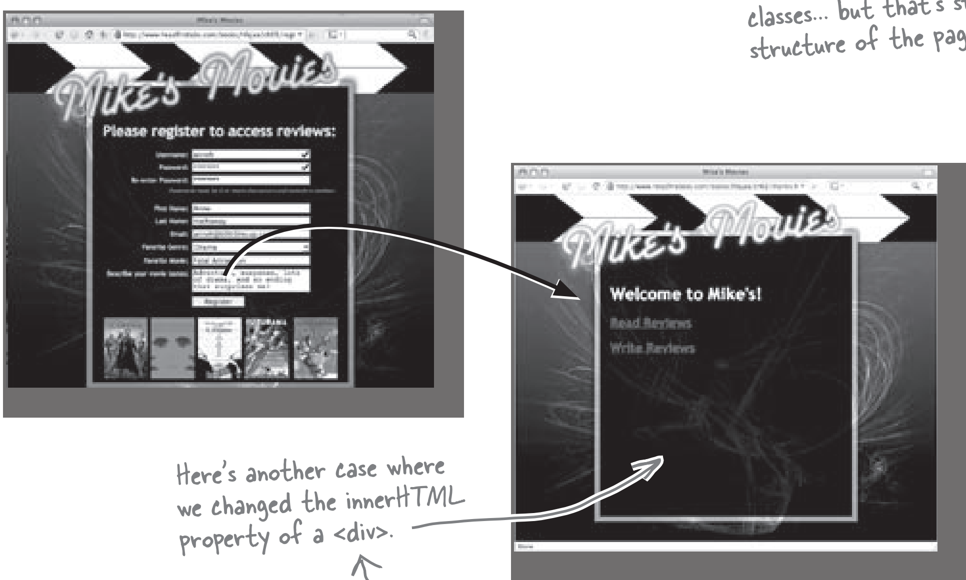
So far, most of the apps we've built have sent requests, gotten a response, and then used that response to update part of a page's content.



In this app, we changed the contents of this <div>.

The structure of this page didn't change, even though the content did.

We also swapped out a couple of CSS classes... but that's still within the existing structure of the page's XHTML.



Here's another case where we changed the innerHTML property of a <div>.

That's the content changing, not the structure of the page.

...or you can change the STRUCTURE of a page

But what if you need to do more than just change the content of a `<div>` or replace the label on a button? What if an image needs to actually move on a page? How would you accomplish that?

You can't change the structure of a page, duh! Why would anyone bother writing XHTML in the first place if someone could just move things around?



Your users can't change your XHTML.

The structure of your page is defined in your XHTML, and people viewing your pages definitely can't mess around with that structure. Otherwise, all the work you'd put into your pages would be a total waste of time.

But what about the **browser**? You'd think it would be okay if the browser moved stuff around... because we can control that with our JavaScript, right?

The browser CAN change your web page's structure



You've already seen that the browser lets you interact with a server-side program, grab elements from a page, and even change properties of those elements. So what about the structure of a page?

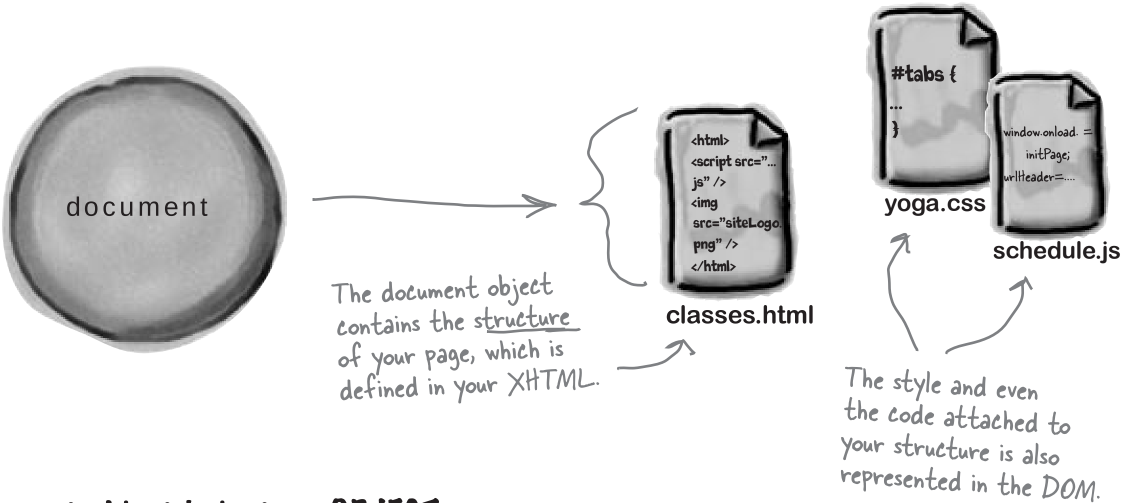
Well, the browser can change that, too. In fact, think about it like this: in a lot of ways, the structure of your page is just a property of the page itself. And you already know how to change an object's properties...

Browsers use the Document Object Model to represent your page

Most people call this the DOM for short.

The browser doesn't see your XHTML as a text file with a bunch of letters and angle brackets. It sees your page as a set of objects, using something called the **Document Object Model**, or DOM.

And everything in the DOM begins with the `document` object. That object represents the very "top level" of your page:



The document object is just an OBJECT

You've actually used the DOM, and in particular the `document` object, several times. Every time you look up an element, you use `document`:

```
var tabs =
  document.getElementById("tabs").getElementsByTagName("a");
```



The document object

getElementById() is a method of the document object.

In fact, every time you treat an element on a page like an object and set properties of that object, you're working with the DOM. That's because the browser uses the DOM to represent every part of your web page.

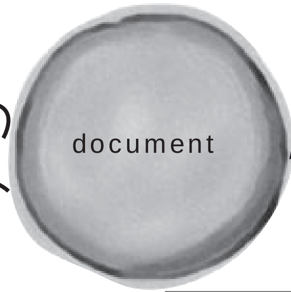
```
currentTab.onmouseover = showHint;
currentTab.onmouseout = hideHint;
currentTab.onclick = showTab;
```

It's the DOM that lets browsers work with parts of a page as JavaScript objects with properties.



The document object... Up Close

Everything in the web browser's model of your web page can be accessed using the JavaScript document object. You've already seen the getElementById() and getElementsByTagName() methods, but there's a lot more that you can do with the document object.



Find an element by its "id" attribute

You've already seen how **getElementByID()** makes it a piece of cake to find an element in your web page using the element's `id` attribute:

```
var tabElement =
  document.getElementById("tabs").value;
```

This gets the element with an id of "tabs."

Get the root element of a document

You can grab the **`<html>`** root element from an XHTML document using the `documentElement` property:

```
var htmlElement = document.documentElement;
```

The root element in an XHTML file is always <html>

Create new parts of a page

You can use various **create...** methods on the `document` object to add elements and text to your page:

```
var myImage = document.createElement("img");

var favShow = document.createTextNode("Bones");
```

This creates a new <img> element.

You can create text and add it anywhere on your page.

Find nodes by their tag type

If you want all the elements of a certain type, for example, all the images, you can use **getElementsByTagName()**. This returns an array, so you'll need to loop through the array to get a particular element:

```
var allDivs =
  document.getElementsByTagName("div");

var firstPara =
  document.getElementsByTagName("p") [0];
```

This returns all the <div> elements.

Get all the <p> elements... ...and return just the first one in the array.

Here's the XHTML that you write...

When you're creating a web page, you write XHTML to represent the structure and content of your page. Then you give that XHTML to the browser, and the browser figures out how to represent the XHTML on the screen. But if you want to change your web page using JavaScript, you need to know exactly how the ***browser*** sees your XHTML.

Suppose you've got this simple XHTML document:

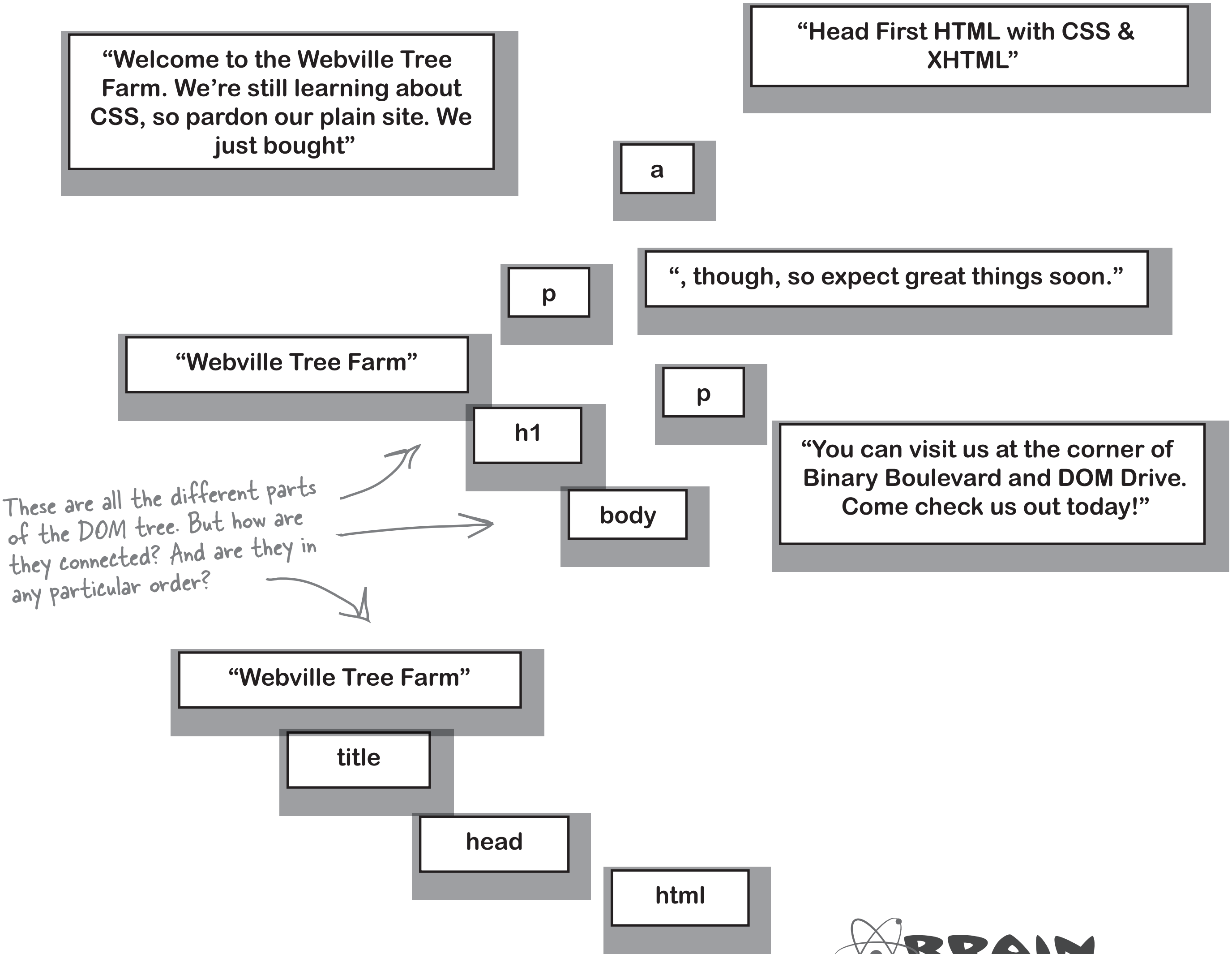
```
<html>
  <head>
    <title>Webville Tree Farm</title>
  </head>
  <body>
    <h1>Webville Tree Farm</h1>
    <p>Welcome to the Webville Tree Farm. We're still learning
    about CSS, so pardon our plain site. We just bought
    <a href="http://www.headfirstlabs.com/books/hfhtml/">
    Head First HTML with CSS &amp; XHTML</a>, though, so expect
    great things soon.</p>
    <p>You can visit us at the corner of Binary Boulevard and
    DOM Drive. Come check us out today!</p>
  </body>
</html>
```

All the bolded tags define the structure of your document.

The text between the tags is the content of your document.

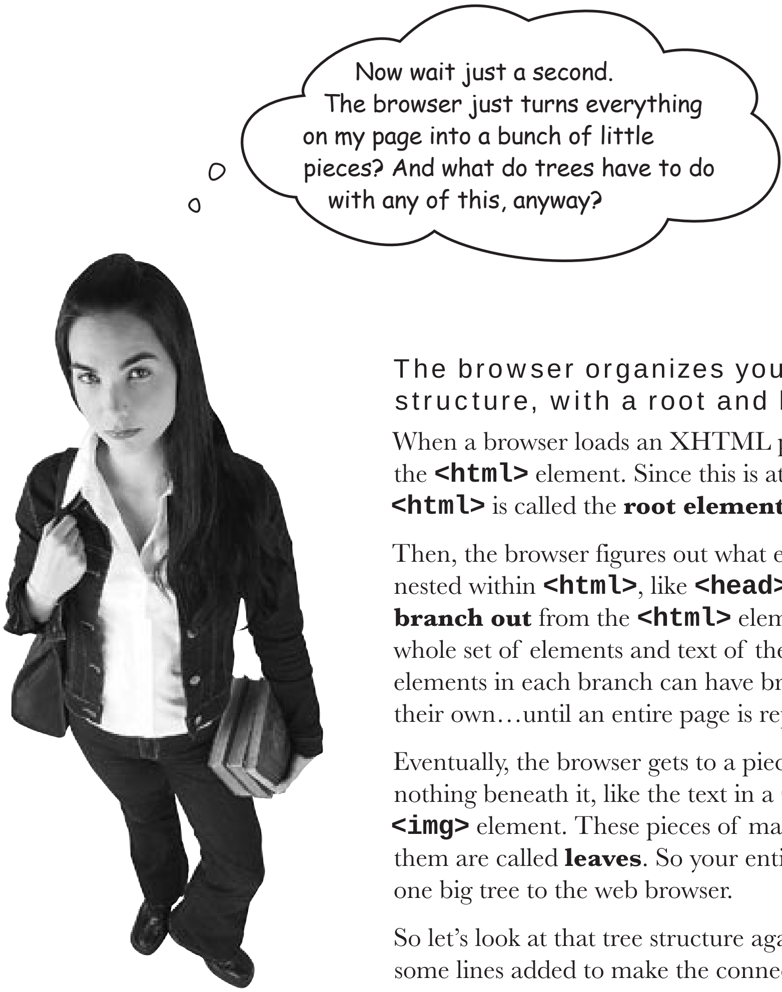
...and here's what your browser sees

The browser has to make some sense of all that markup, and organize it in a way that allows the browser—and your JavaScript code—to work with the page. So the browser turns your XHTML page into a tree of objects:



BRAIN POWER

How would ***YOU*** order and relate these different parts of the DOM tree?



The browser organizes your page into a tree structure, with a root and branches.

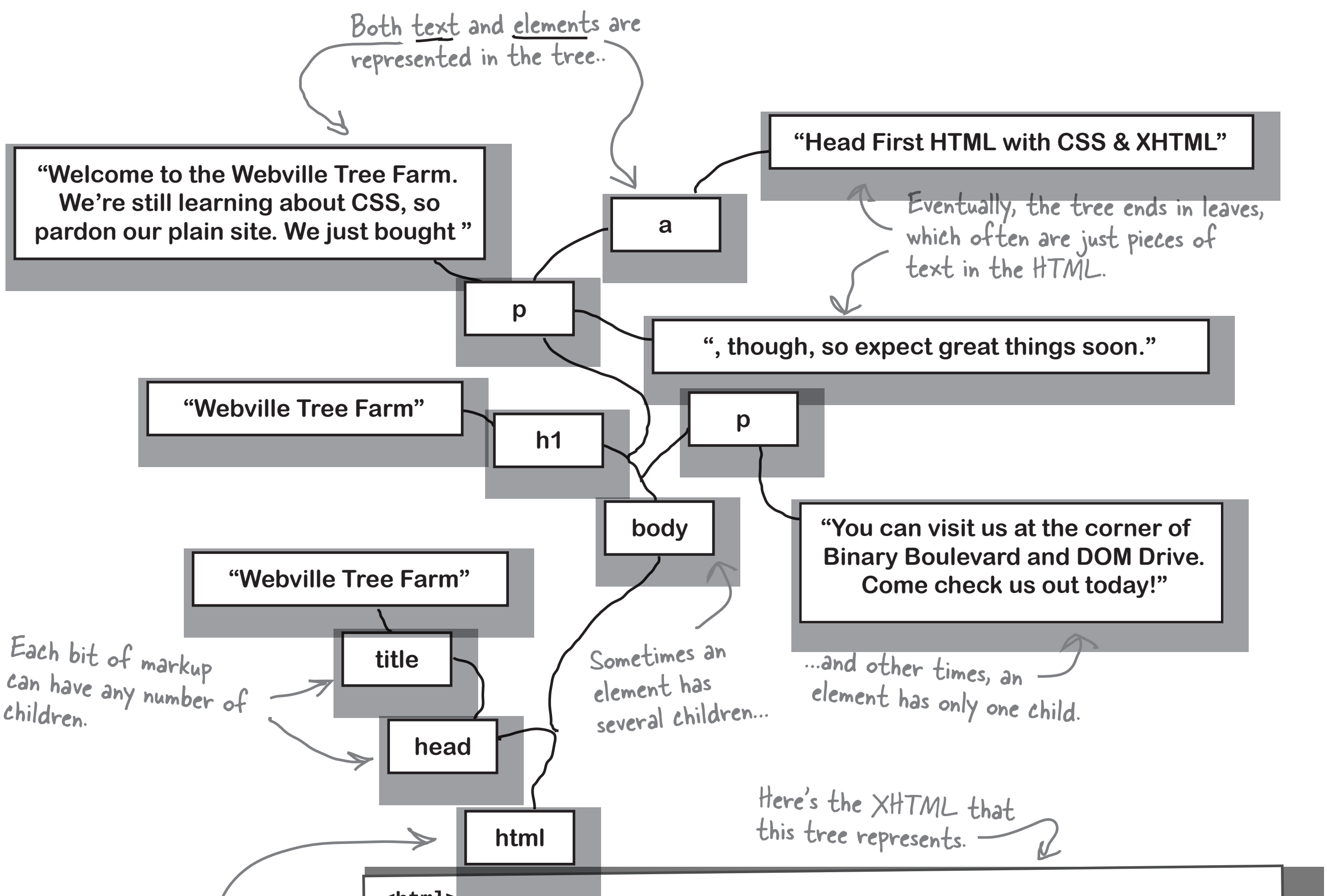
When a browser loads an XHTML page, it starts out with the **`<html>`** element. Since this is at the "root" of the page, **`<html>`** is called the **root element**.

Then, the browser figures out what elements are directly nested within **`<html>`**, like **`<head>`** and **`<body>`**. These **branch out** from the **`<html>`** element, and they have a whole set of elements and text of their own. Of course, the elements in each branch can have branches and children of their own…until an entire page is represented.

Eventually, the browser gets to a piece of markup that has nothing beneath it, like the text in a **`<p>`** element or an **`<img>`** element. These pieces of markup with nothing under them are called **leaves**. So your entire page ends up being one big tree to the web browser.

So let's look at that tree structure again, but this time, with some lines added to make the connections between the markup a little clearer.

Your page is a set of related objects




```
<html>
  <head>
    <title>Webville Tree Farm</title>
  </head>
  <body>
    <h1>Webville Tree Farm</h1>
    <p>Welcome to the Webvill Tree Farm. We're still learning
    about CSS, so pardon our plain site. We just bought
    <a href="http://www.headfirstlabs.com/books/hfhtml/">
    Head First HTML with CSS &amp; XHTML</
    a>, though, so expect great things soon.</p>
    <p>You can visit us at the corner of Binary Boulevard and DOM
    Drive. Come check us out today!</>
  </body>
</html>
```

there are no Dumb Questions

Q: Do I need to use a request object to write JavaScript code that uses the DOM?

A: Nope. The browser handles creation of the DOM tree and exposes all the methods of the document object automatically. In fact, even when you're not writing JavaScript at all, browsers still use the DOM to represent your page.

Q: So if the DOM doesn't use a request object, is it really part of Ajax?

A: Depends who you talk to. Ajax is really just a way of thinking about web pages, and a whole slew of other technologies, that helps you achieve interactive pages in really usable ways. So the DOM is definitely a part of that. You'll use the DOM a lot in the next couple of chapters to build interactive and usable apps.

Q: What about all that DOM Level 0 and DOM Level 2 stuff? Am I going to have trouble with Internet Explorer again?

A: All modern browsers are compatible with the World Wide Web Consortium's (W3C) DOM specification, but the specification leaves some decisions up to the browser designer. The designers of IE made a different decision about how to build the DOM tree than a lot of the designers of other major browsers. But it's not a big problem, and with a few more utility functions, your code will work on ***all*** major browsers, including Internet Explorer.

Q: It looks like you called some parts of the markup "children." So an element can have "child elements"?

A: Yes. When the browser organizes your XHTML into a tree, it begins with the root element, the element that surrounds everything else. Then, that element has an element within it, like `<head>` or `<body>`. Those can be called *nested elements*, but in DOM, they're called ***child elements***.

In fact, you can think of the DOM tree as a family tree, with family terms applying everywhere. For example, the `<head>` element is the parent of the `<title>` element, and most `<a>` elements have children: the text label for the link.

Q: You're throwing a bunch of new terms around. How am I supposed to keep up with all of this?

A: It's not as hard as it seems. Just keep the idea of a family tree in mind, and you shouldn't have any trouble. You've been using terms like root, branch, and leaf for years. As for parent and child, anytime you move away from the root, you're moving towards a child. The only term that may be totally new to you is "node," and we're just about to take a look at that…

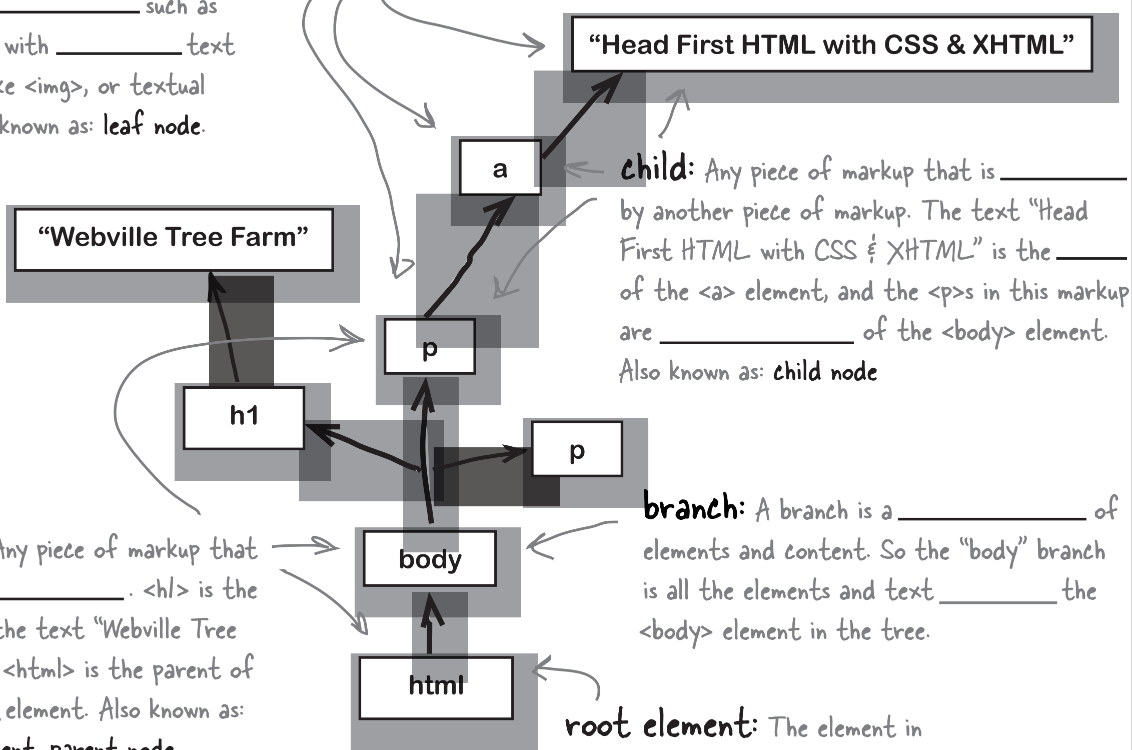
Exercise

Write Your Own Web Dictionary

What good is looking at a bunch of definitions? This is Head First, and we want your brain working, not just your eyes. Below are several entries from a Web Dictionary, with some of the words in each definition removed. Your job is to complete each entry by filling in the blanks.

node: Any _____ piece of markup, such as an element or text. The <a> element is an _____ node, while the "Head First HTML with CSS & XHTML" text is a _____ node.

leaf: A piece of markup that has _____ such as an element with _____ text content, like <img>, or textual data. Also known as: **leaf node**.



child: Any piece of markup that is _____ by another piece of markup. The text "Head First HTML with CSS & XHTML" is the _____ of the <a> element, and the <p>s in this markup are _____ of the <body> element. Also known as: **child node**

parent: Any piece of markup that contains _____. <h1> is the parent of the text "Webville Tree Farm," and <html> is the parent of the _____ element. Also known as: **parent element, parent node**.

branch: A branch is a _____ of elements and content. So the "body" branch is all the elements and text _____ the <body> element in the tree.

root element: The element in a _____ that _____ all other elements. In XHTML, the root element is always _____.

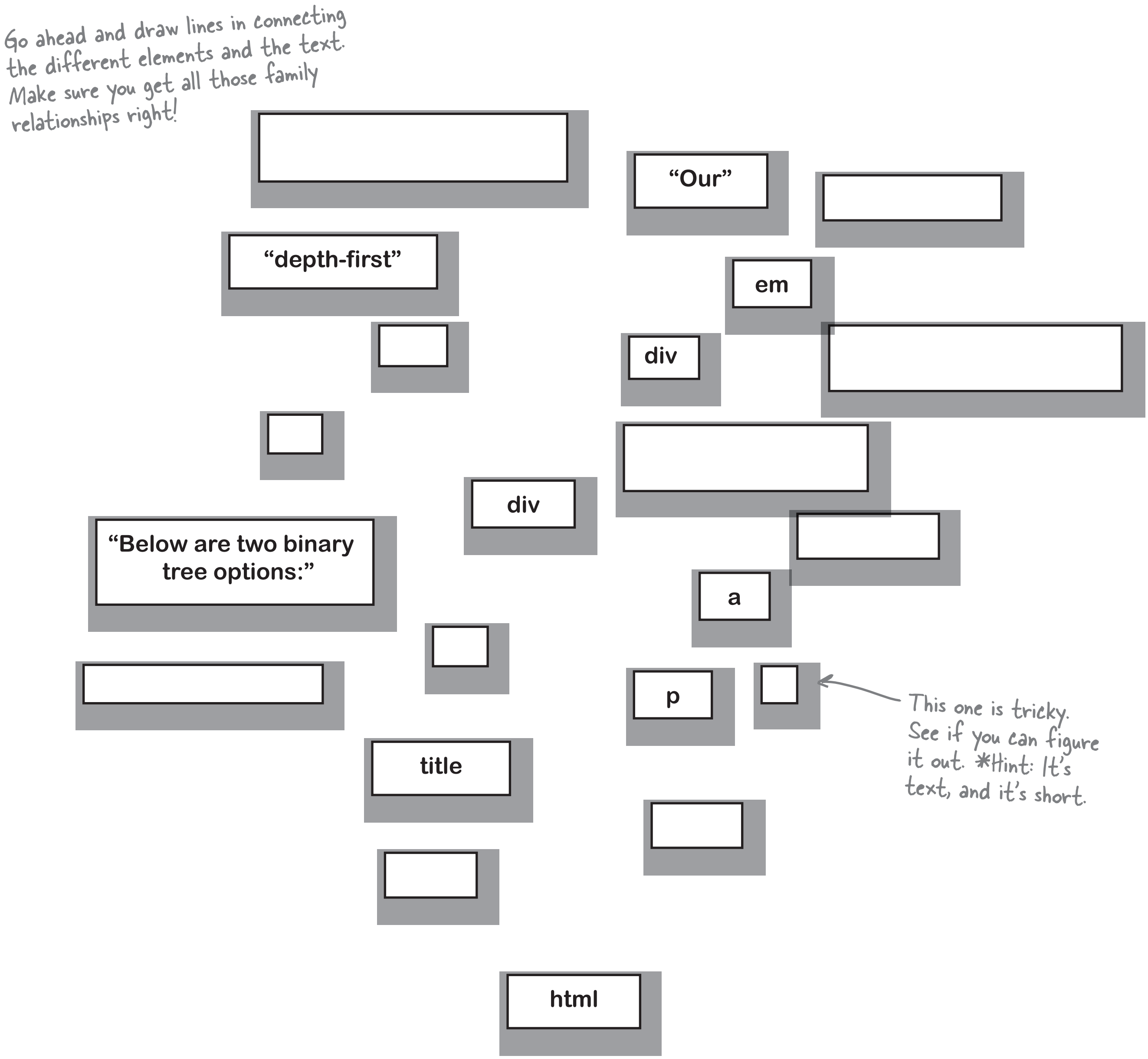
no children	child	children	under	contained
other markup	text	collection	single	document
element	contains	<body>	no	<html>

Here are the words you should use to fill in the blanks.

Sharpen your pencil

It's time to load markup trees into your brain. Below is an XHTML document. Your job is to figure out how a web browser organizes this markup into a tree structure. On the right is the tree, ready for you to fill in its branches and the relationships between each piece. To get you started, we've provided spaces for each piece of markup; be sure you've filled each space with an element or text from the XHTML markup before showing off your DOM tree to anyone else!

```
<html>
  <head>
    <title>Binary Tree Selection</title>
  </head>
  <body>
    <p>Below are two binary tree options:</p>
    <div>
    Our <em>depth-first</em> trees are great for folks who
    are far away.
    </div>
    <div>
    Our <em>breadth-first</em> trees are a favorite for
    nearby neighbors.
    </div>
    <p>You can view other products in the
      <a href="menu.html">Main Menu</a>. </p>
  </body>
</html>
```



Answers on page 243.

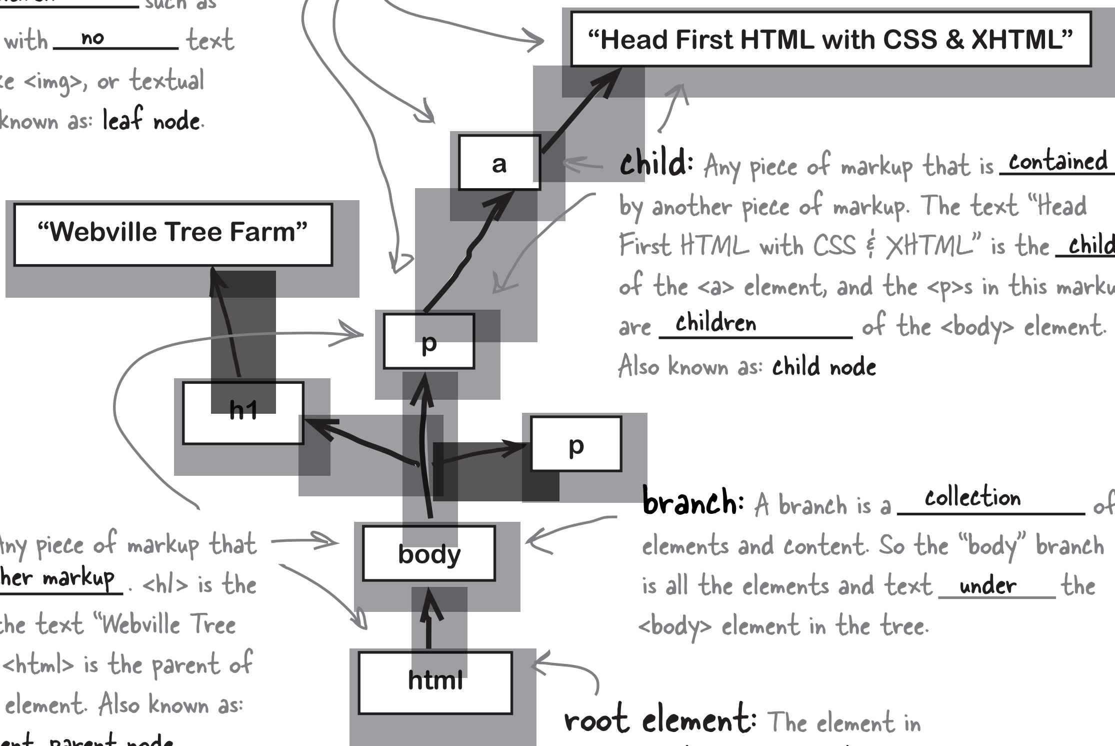
Exercise Solution

Write Your Own Web Dictionary

Below are several entries from a Web Dictionary, with some of the words in each definition removed. Your job was to complete each entry by filling in the blanks.

node: Any **single** piece of markup, such as an element or text. The <a> element is an **element** node, while the "Head First HTML with CSS & XHTML" text is a **text** node.

leaf: A piece of markup that has **no children** such as an element with **no** text content, like <img>, or textual data. Also known as: **leaf node**.



child: Any piece of markup that is **contained** by another piece of markup. The text "Head First HTML with CSS & XHTML" is the **child** of the <a> element, and the <p>s in this markup are **children** of the <body> element. Also known as: **child node**

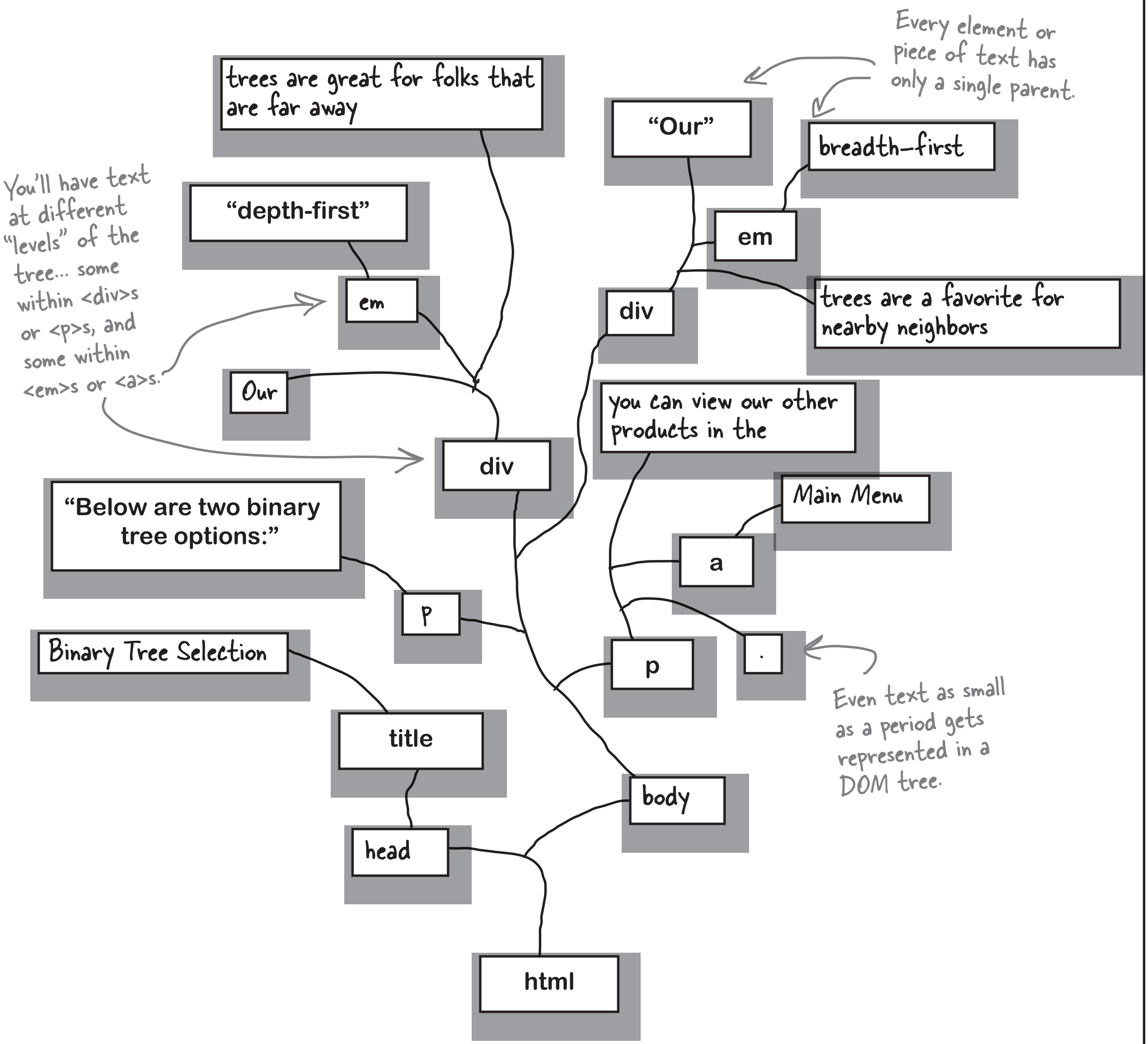
parent: Any piece of markup that contains **other markup**. <h1> is the parent of the text "Webville Tree Farm", and <html> is the parent of the **<body>** element. Also known as: **parent element, parent node**.

branch: A branch is a **collection** of elements and content. So the "body" branch is all the elements and text **under** the <body> element in the tree.

root element: The element in a **document** that **contains** all other elements. In XHTML, the root element is always **<html>**.

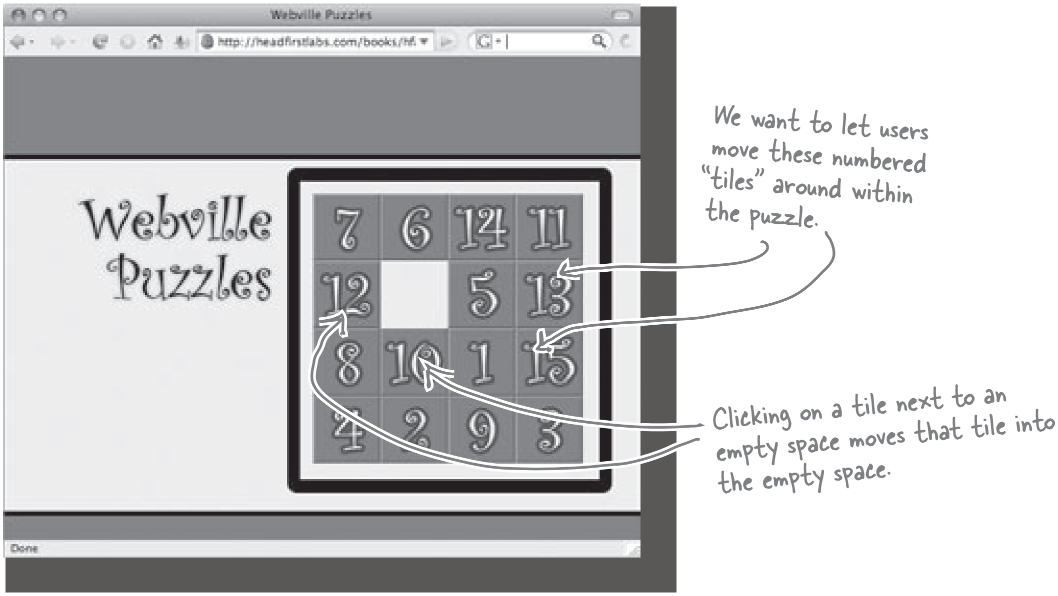
Sharpen your pencil Solution

Your job was to build a DOM tree from the XHTML on page 240. You also should have drawn in the connections between the different elements and text. How did you do?



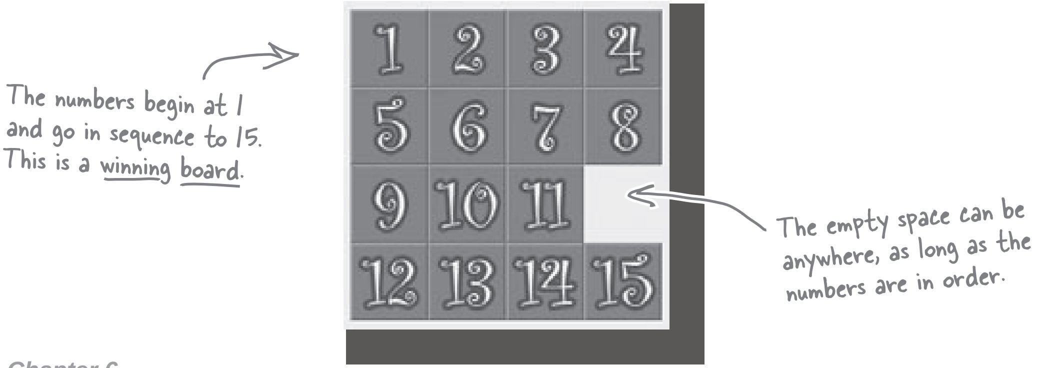
Let's use the DOM to build a dynamic app

Now that we know a bit about the DOM, we can use that knowledge to make our apps do even more interesting things. Let's take on a project for the Webville Puzzle Company. They've been working on a bunch of new web-based games, and they need help with their online Fifteen Puzzle.



We're not replacing content, we're moving it

In a Fifteen Puzzle, you can **move** a tile into the empty space, which then creates a new empty space. Then you can **move** another tile into *that* new empty space, and so on. There's always one empty space, and the goal is to get all the numbers lined up in sequential order, like this:





We need to move those tiles around... and that requires the DOM.

This is a perfect example of needing the DOM. We don't want to just change the content of a table, or replace some text on a button or in a `<p>`. Instead, we need to move around the images that represent a tile.

Webville Puzzles is using a table with four rows and four columns to represent their board. So we might need to move an image in the third row, fourth column to the empty space in the third row, third column. We can't just change the `innerHTML` property of a `<div>` or `<td>` to get that working.

What we need is a way to actually grab an `<img>`, and move it within the overall table. And that's where the DOM comes in handy. And, as you'll soon see, this is *exactly* the sort of thing that Ajax apps have to do all the time: dynamically change a page.

All Ajax apps need to respond DYNAMICALLY to users.

The DOM lets you CHANGE a page without reloading that page.

BRAIN POWER

What specific steps do you think you'll have to take to move an <img> from one cell of a table to a different cell?

You start with XHTML...

To really understand how the DOM helps out, let's take a look at Webville Puzzles' XHTML, and see what the browser does with that XHTML. Then we can figure out how to use the DOM to make the page do what ***we*** want.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Webville Puzzles</title>
  <link rel="stylesheet" href="css/puzzle.css" type="text/css" />
  <script src="scripts/fifteen.js" type="text/javascript"></script>
</head>
<body>
 <div id="puzzle">
  <h1 id="logo">Webville Puzzles</h1>
  <div id="puzzleGrid">
   <table cellspacing="0" cellpadding="0">
     <tr>
      <td id="cell11">
        <img src="images/07.png" alt="7" width="69" height="69" />
      </td>
      <td id="cell12">
        <img src="images/06.png" alt="6" width="69" height="69" />
      </td>
      <td id="cell13">
        <img src="images/14.png" alt="14" width="69" height="69" />
      </td>
      <td id="cell14">
        <img src="images/11.png" alt="11" width="69" height="69" />
      </td>
     </tr>
     <tr>
      <td id="cell21">
        <img src="images/12.png" alt="12" width="69" height="69" />
      </td>
      <td id="cell22">
        <img src="images/empty.png" alt="empty" width="69" height="69" />
      </td>
      <td id="cell23">
        <img src="images/05.png" alt="5" width="69" height="69" />
      </td>
      <td id="cell24">
        <img src="images/13.png" alt="13" width="69" height="69" />
      </td>
     </tr>
    ... etc ...
   </table>
  </div>
 </div>
</body>
</html>
```

There's no JavaScript yet, but we'll need some soon. Go ahead and add a reference to fifteen.js, which we'll build throughout this chapter.

The puzzle is represented by a <table> element.

Each table cell is labeled with an id.

Each tile is a single <img> within a single table cell.

The empty tile is also an image.

The XHTML for the puzzle is in fifteen-puzzle.html. You can download the source from the Head First Labs website.



fifteen-puzzle.html

Sharpen your pencil

Go ahead and draw what you think the DOM tree for fifteen-puzzle.html looks like. This time, though, you don't have to put the root element at the base of the tree. You can put it anywhere you want: the top, the bottom, or to one side.

there are no Dumb Questions

Q: There isn't any request being made in this puzzle, is there?

A: No, at least not right now. The program is all client-side.

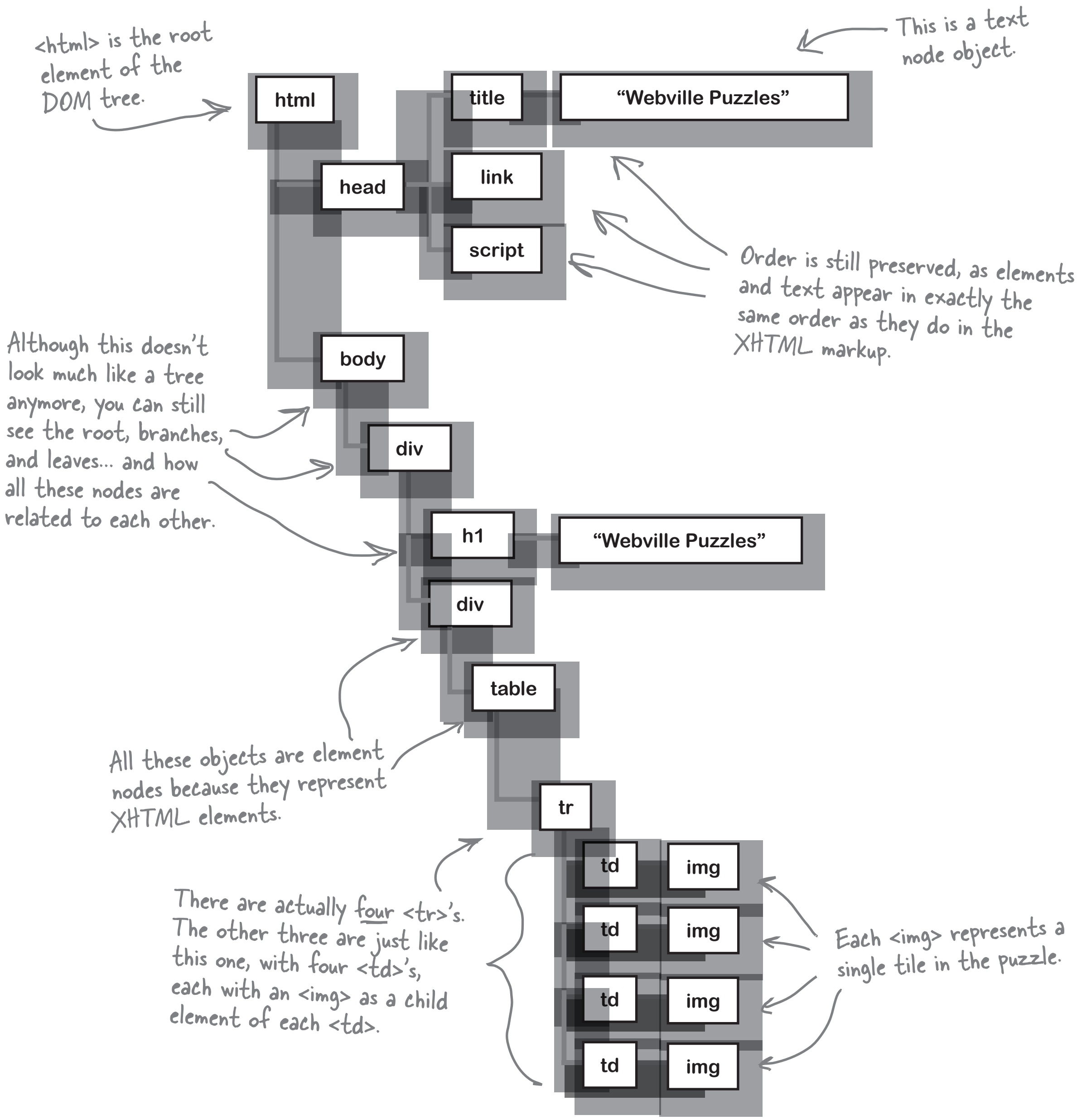
Q: So this isn't Ajax at all, is it?

A: Well, that gets back to the "What is Ajax?" question. If you think Ajax apps are only ones that make requests using `XMLHttpRequest`, then this isn't an Ajax app. But if you think of Ajax apps more as responsive, JavaScript-driven apps that are very usable, then you might decide otherwise.

Either way, this app is really all about controlling the DOM... and that's something that will help your Ajax programming, no matter what you think constitutes an Ajax app.

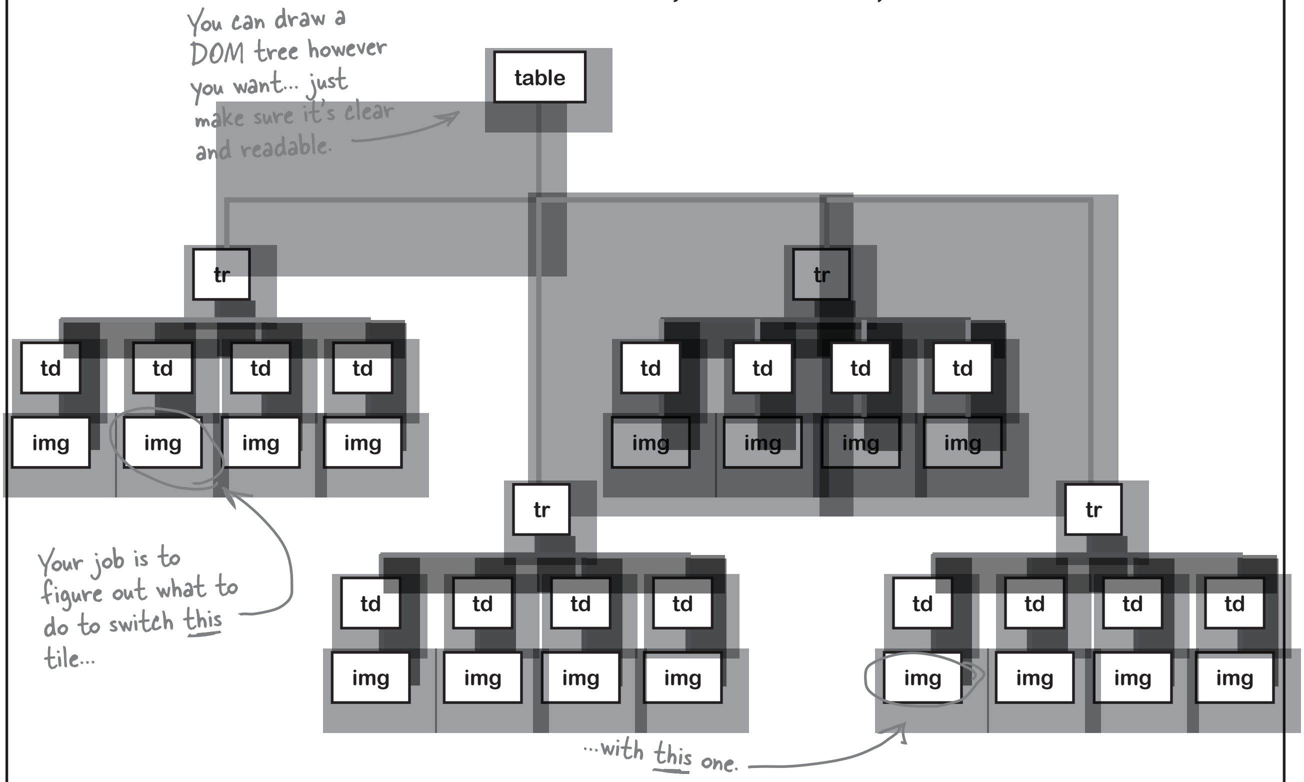
Sharpen your pencil Solution

Your job was to draw out a DOM tree for the fifteen-puzzle.html's XHTML structure and content. Here's what we did... we started with the root element on the top-left, and worked our way down. Did you come up with something similar?



Sharpen your pencil

The exercises just keep coming! This time, you've got to swap two particular nodes in the DOM tree structure below. Your job is to write out exactly what steps you'd take. Don't worry about method names, just write out what you'd do.



Assume you know which ***table cell*** was clicked on, and you also know the destination table cell. What would you do?

1. ....

2. ....

3. ....

4. ....

5. ....

...etc...

You can use as many steps as you need.

You can write things like "Get the first child of the current element" or "Find all the elements named 'img'."

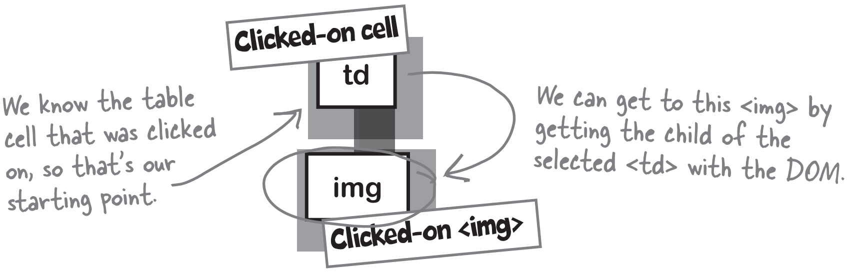
Hint: see if you can use the word "parent" at least once... maybe even twice!

Sharpen your pencil Solution

Your job was to write out exactly what steps you'd take. Assume you know which ***table cell*** was clicked on, and you also know the destination table cell. What would you do?

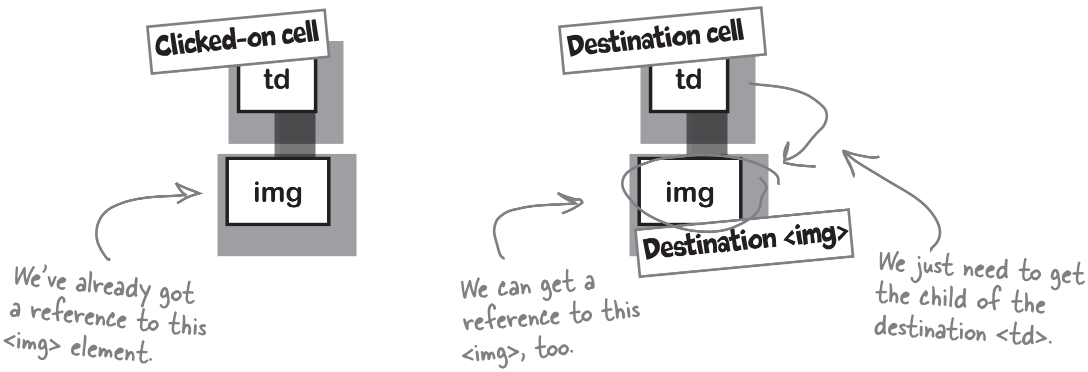
1 Get the child of the selected table cell

You could use `getElementsByTagName()`, but we know that the `<img>` representing the clicked-on tile is the child of the selected cell. So we can use the DOM to get that child element.



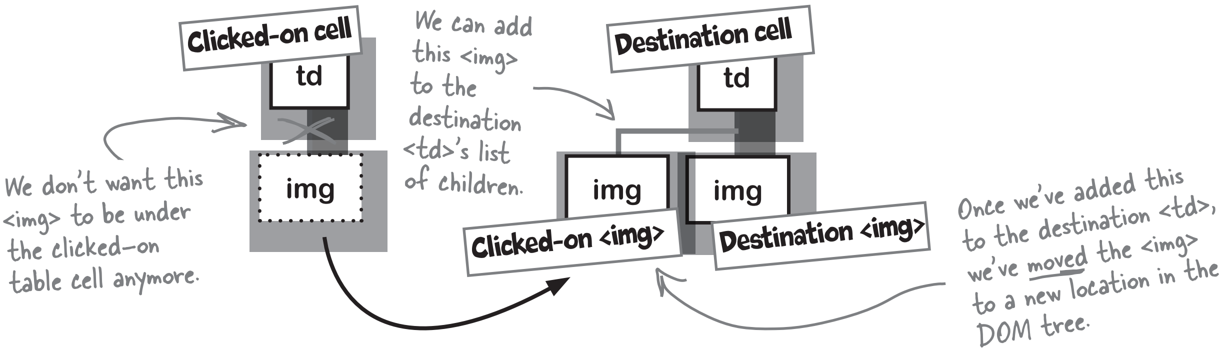
2 Get the child of the destination table cell

Once we start swapping things around, it's going to be harder to keep the selected `<img>` separate from the destination `<img>`. So before we start moving things around, let's get a reference to the `<img>` in the destination `<td>`, too.



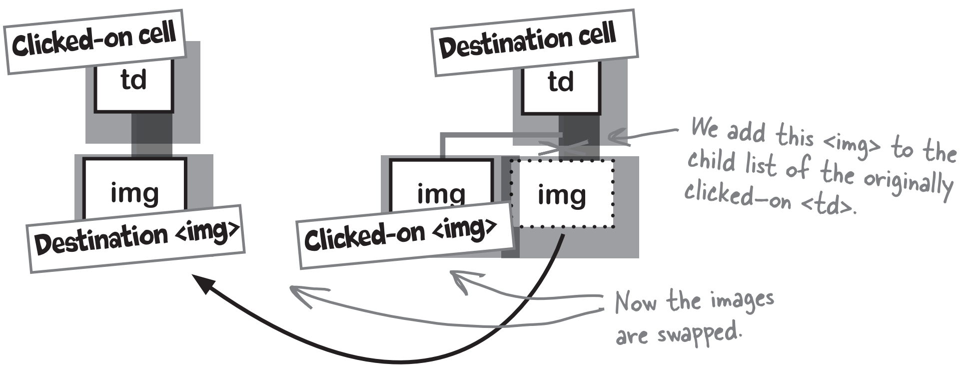
3 Add the clicked on <img> as a child of the new destination table cell.

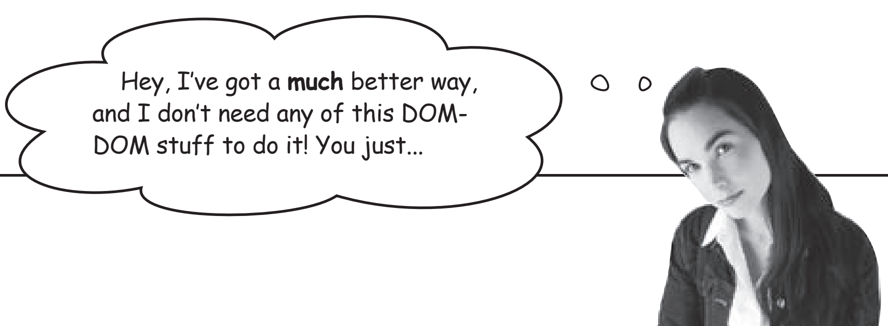
We need to move the `<img>` in the selected cell to the destination cell. So we can just add the clicked-on `<img>` to the destination `<td>`'s list of children.



4 Add the <img> in the destination cell as a child of the originally clicked-on table cell.

Now for the other part of the swap. We need to move the destination `<img>` under the `<td>` that was originally clicked on. Here's why we got the reference to this `<img>` in step 2: since there are two child `<img>`'s under the destination `<td>`, having a reference already makes this easy.







...get the src property of the first <img>, and swap it out with the src property of the second <img>. All you need is a temporary string, and you're done. No DOM, no new syntax. Nice, huh?

Do you want to CHANGE an element or MOVE an element? There's a big difference.

You could definitely write code that simply swaps out the values of the two `<img>`'s src properties, like this:

```
var tmp = selectedImage.src;
selectedImage.src = destinationImage.src;
destinationImage.src = tmp;
```

This code swaps the textual src property of two images.

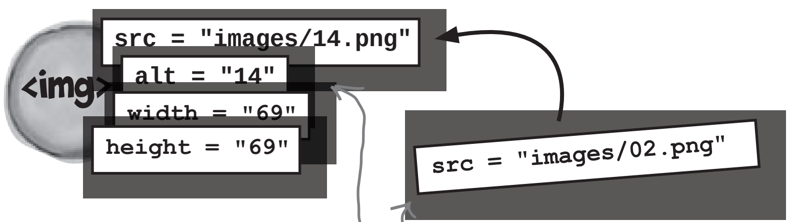
The problem with this is that you're actually just changing the *properties* of an image, and not ***moving*** those images around on the page.

So what's the big deal with that? Well, what about the other properties of each image? Remember that each `<img>` had an `alt` attribute?

```
<img src="images/14.png" alt="14"
     width="69" height="69" />
```

The alt, width, and height are properties of this image, just like its src attribute.

If you change the `src` attribute, you're only changing a part of the `<img>`. The rest would stay the same... and then the alt attribute would not match the image!



If you change the src property, then the image (tile #2) won't match the alt attribute ("14").

What you need to do is ***swap*** the entire `<img>` objects. That way, each `<img>` keeps its properties. The image doesn't change, but the *location* of that `<img>` in the DOM tree (and on the visual representation of the page) does.



JavaScript & DOM Magnets

You've already figured out what needs to happen to swap two tiles. Now it's time to turn those general steps into actual code. Below is the skeleton for a new function, swapTiles(). But all the pieces of code have fallen to the ground... can you figure out how to complete the function?

```
function swapTiles(________________, _______________) {

  _________________ = ________________._______________;

  _________________ = ________________._______________;

  _________________._______________(________________);

  _________________._______________(________________);
}
```

selectedImage
selectedCell
appendChild
selectedImage
destinationImage
destinationCell
selectedCell
destinationCell
appendChild
firstChild
destinationCell
destinationImage
selectedCell
firstChild



JavaScript & DOM Magnet Solutions

It's time to turn those general steps from page 250 into actual code. Below is the skeleton for a new function, swapTiles(). Your job was to put the pieces of code into a working function.

```
function swapTiles( selectedCell , destinationCell ) {
    selectedImage = selectedCell . firstChild ;
    destinationImage = destinationCell . firstChild ;
    selectedCell . appendChild ( destinationImage );
    destinationCell . appendChild ( selectedImage );
}
```

This gets a reference to the two images to swap...

...and this swaps the two images.

What about all that parent stuff? Is there not a parent property for each element, or node, or whatever? We could set the parent property of selectedImage to destinationCell, and vice versa for the destinationImage.



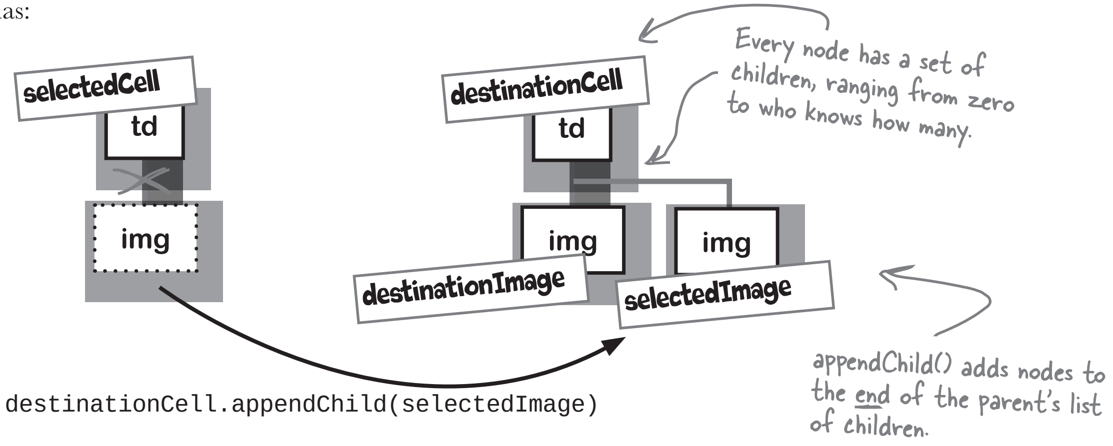
Every node has a parentNode property... but the parentNode property is read-only.

Every node in a DOM tree—that's elements, text, even attributes—has a property called `parentNode`. That property gives you the ***parent*** of the current node. So for example, the parent node of an `<img>` in a table cell is the enclosing `<td>`.

But in the DOM, that's a *read-only* property. So you can get the parent of a node, but you can't set the parent. Instead, you have to use a method like `appendChild()`.

appendChild() adds a new child to a node

`appendChild()` is a method used to add a new child node to an element. So if you run `destinationCell.appendChild(selectedImage)`, you're adding the `selectedImage` node to the children that `destinationCell` already has:



A new child gets a new parent... automatically

When you assign a node a new child, that new child's `parentNode` property is *automatically updated*. So even though you can't change the `parentNode` property directly, you can move a node, and let the DOM and your browser handle changing the property for you.

there are no Dumb Questions

Q: So I can use all the DOM methods from my JavaScript automatically?

A: That's mostly right. There are a few exceptions that we'll look at soon, but for the most part, the DOM is yours to use from any JavaScript code you're writing.

Q: And a DOM tree is made up of nodes, like elements and text, right?

A: Right, but don't forget about attributes, too. A node is pretty much anything that can appear on a page, but the most common nodes are elements, attributes, and text.

Q: And a node has a parent and children?

A: All nodes have parents, but not all nodes have children. Text and attribute nodes have no children, and an empty element with no content has no children.

Q: What's the parent of the root element?

A: The `document` object. That's why you can use the `document` object to find anything in your web page.

Q: Are there other methods to add child nodes like appendChild()?

A: There sure are. We'll be looking at lots of those in the next chapter.

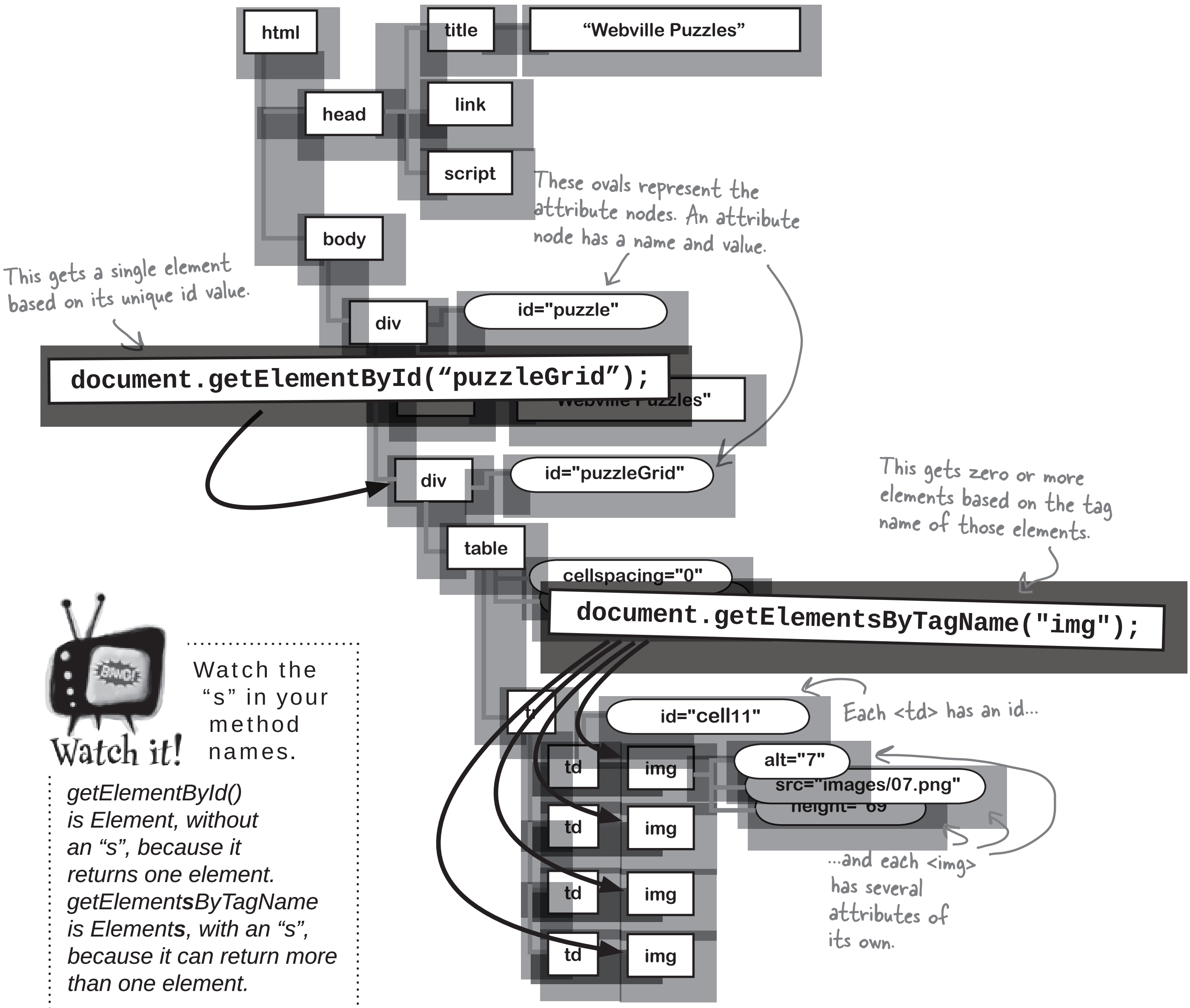
Q: Why is this better than changing out the src property of an <img>?

A: Because you don't want to *modify* the image displayed; you want to *move* that image to a new cell. If you wanted an image to stay the same, with its `alt` tag and `height` and `title`, you'd change the `src` property. But we want to move the image, so we use the DOM.

You can locate elements by name or by id

If you think about your page as a collection of nodes in a DOM tree, methods like `getElementById()` and `getElementsByTagName()` make a lot more sense.

You use `getElementById()` to find a *specific* node anywhere in the tree, using the node's `id`. And `getElementsByTagName()` finds *all* elements in the tree, based on the node's tag name.



Watch it!

Watch the "s" in your method names.

*getElementById() is Element, without an "s", because it returns one element. getElement**s**ByTagName is Element**s**, with an "s", because it can return more than one element.*

Exercise

Go ahead and write the code for an initPage() function. You need to make sure that every time a table cell is clicked on, an event handler called tileClick() gets run. We'll write the code for tileClick() later, but you may want to build a test version with an alert() statement to make sure your code works before turning the page.

Answers on the next page.

Sharpen your pencil

Here are a few questions to get your left brain into gear. Answer each before turning the page... and once you're done, you might want to double-check your code for initPage() above, too.

1. Should the event handler for moving a tile be on the table cell or the image within that cell?

☐ table cell (<td>) ☐ image (<img>)

2. Why did you make the choice you did? ....

....

3. How can we figure out if an empty tile was clicked on? ....

....

4. How can we figure out the destination cell for a tile? ....

....

Answers on page 261.

Exercise Solution

Your job was to write an initPage() function that set up the event handlers for the Fifteen Puzzle. What did you come up with? Here's what we did:

```
window.onload = initPage;

function initPage() {
  var table = document.getElementById("puzzleGrid");
  var cells = table.getElementsByTagName("td");
  for (var i=0; i<cells.length; i++) {
    var cell = cells[i];
    cell.onclick = tileClick;
  }
}

function tileClick() {
  alert("You clicked me!");
}
```

Remember to assign the initPage() function to the window.onload event.

First, we locate the <div> with the table and cells we want to attach handlers to.

We want every <td> in that table.

For each cell...

...assign tileClick() to the onclick event.

We built a simple event handler to test things out.

Test Drive

Add initPage(), tileClick(), and swapTiles() to a script called fifteen. js. Be sure you reference the file in your XHTML, and try out the Fifteen Puzzle with the event handlers on each table cell.





Table Cells Exposed

This week's interview:
Interview with a new parent

Head First: So I hear you're a new parent, <td>?

<td>: That's right. I've got a sweet little <img> to call my own.

Head First: So is this your first child?

<td>: Well, it depends on who you ask. Some browsers say that <img> is my first child, but others think I've got a lot of empty children floating around.

Head First: Empty children?

<td>: Yup. You know, empty spaces, carriage returns. It's nothing to worry about.

Head First: Nothing to worry about? That sounds pretty serious... you might have more children, and that's no big deal?

<td>: Relax, it's all in how you handle it. Most people just skip over all those nothings to get to my flashy little <img>.

Head First: This is all pretty confusing. Do you think our audience really understands what you're talking about?

<td>: If they don't know, I'll bet they will soon. Just wait and see.

Head First: Well... hmmm... I guess... I guess that's all for now. Hopefully we'll make some sense of all this, and get back to you soon, faithful listeners.

there are no Dumb Questions

Q: Why do you have the puzzleGrid id on a <div>, and not on the <table> itself?

A: DOM Level 2 browsers and Internet Explorer handle tables, and CSS styles applied to those tables, pretty differently. The easiest way to get a page with a table looking similar on IE and Firefox, Safari, etc., is to style a `<div>` surrounding a `<table>`, instead of the `<table>` itself.

Since it's easiest to style an element with an `id`, we put the puzzleGrid id on the `<div>` we wanted to style: the one surrounding the `<table>` cell.

Q: So that's why you used getElementById() to find that <div>, and not the actual <table>?

A: Right. We could have put an id on the `<table>`, too, but it's not really necessary. The only thing in the puzzleGrid `<div>` is the table we want, along with all those clickable cells. So it was easier to just find the `<div>`, and then find all the `<td>`'s within that.

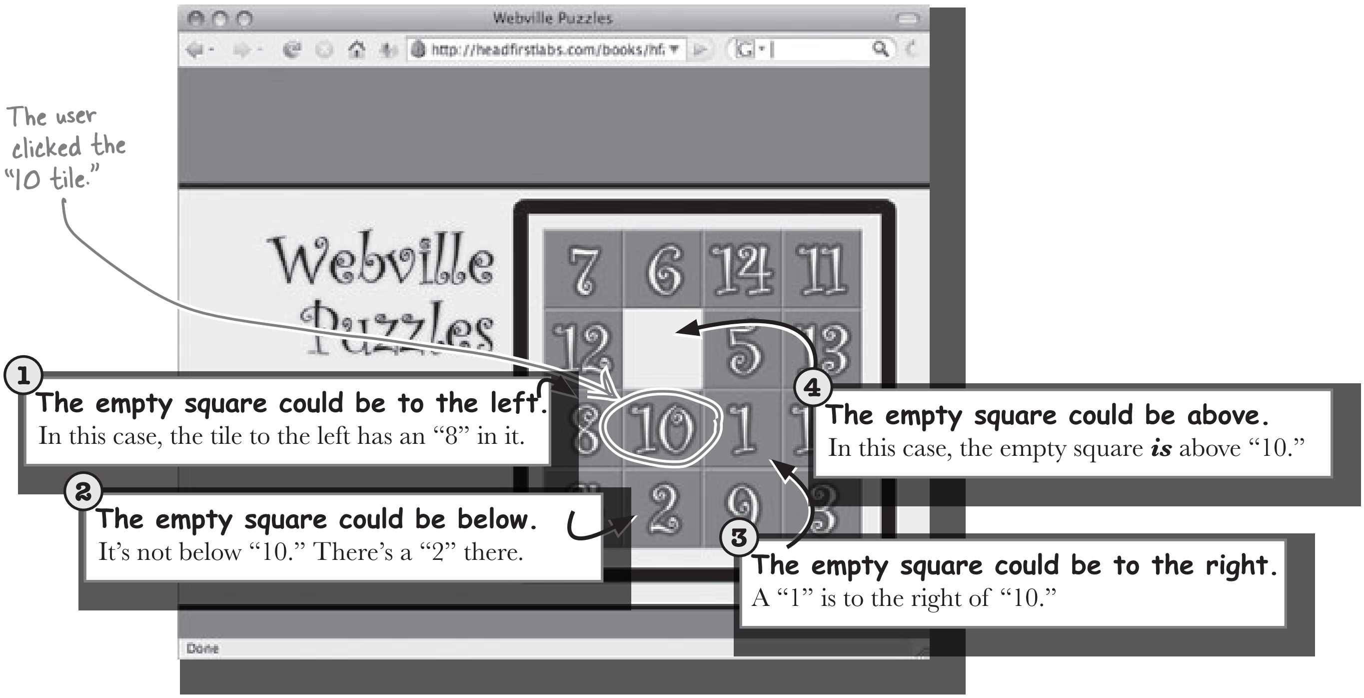
BRAIN POWER

What do you think <td> is talking about in the interview above? Are there any functions we've written or will write that might need to worry about those "nothing" children that <td> mentioned?

Can I move the clicked tile?

Now that the basic structure is in place, it's time to get the puzzle working. Since a tile can only be moved to the empty square, the first thing we need to figure out is, "Where's the empty square?"

For any clicked-on tile, there are six different possibilities for where the empty tile is. Suppose the user clicked the "10" tile on the board below:



Sharpen your pencil

There are two more possible situations related to the position of the empty tile. Can you figure out what they are?

5 ....
....

6 ....
....

Answers on page 265.

Sharpen your pencil Solution

Here are a few questions to get your left brain into gear. Answer each before turning the page... and once you're done, you might want to double-check your code for initPage() above, too.

1. Should the event handler for moving a tile be on the table cell or the image within that cell?

☐ table cell (<td>) ☐ image (<img>)

2. Why did you make the choice you did?
3. How can we figure out if an empty tile was clicked on?
4. How can we figure out the destination cell for a tile?

Well, first of all, I think we should put the event handler on the table cell, not on the image itself.



Joe: Why? The user's clicking on "7," not the second tile on the third row.

Frank: Well, they're clicking on the table cell that image is in, too.

Jill: So suppose we put the handler on the image. And then when a user clicks on the image...

Joe: ...we swap that image out with the empty square...

Jill: Right. But the handler's attached to the ***image***, not the table cell.

Frank: Oh. I see.

Joe: What? I don't get it.

Frank: The event handler would move with the image. So every time an image gets moved, the event handler moves with it.

Joe: So?

Frank: Well, we're going to use the DOM to figure out where the empty square is in relation to the clicked-on image, right?

Joe: I guess so. What's that got to do with the handler on the image?

Frank: If the handler's on the image, we'll constantly have to be getting the image's parent. If the handler's on the cell, we can avoid that extra step. We can just check the cells around the clicked-on cell.

Jill: Exactly! We don't need to move to the image's parent cell in our handler.

Joe: So all this is to avoid one line of code? Just asking the image for its parent?

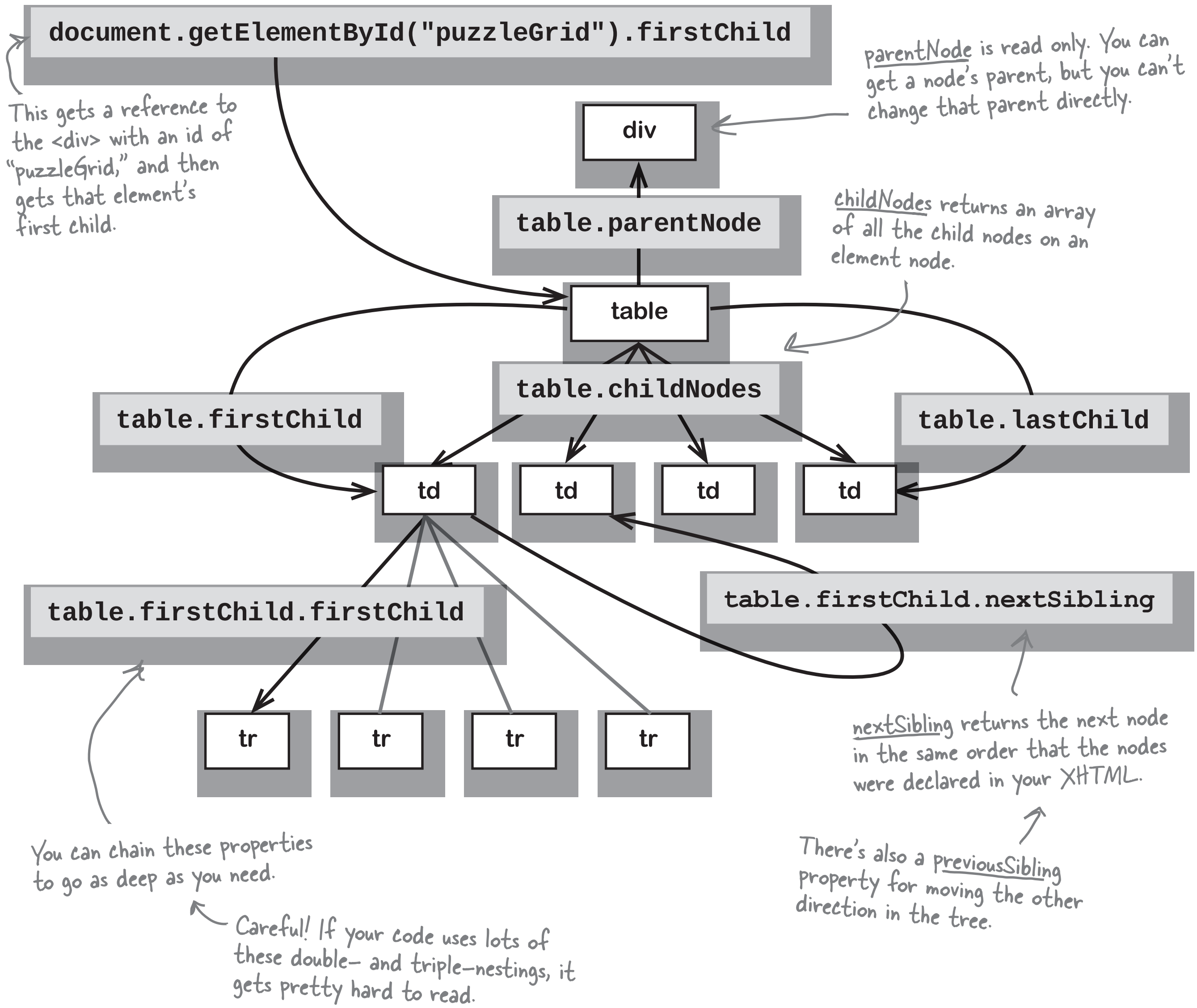
Jill: One line of code ***for every click***. That could be hundreds of clicks... or even thousands! Have you ever worked one of those puzzles? It takes some time, you know.

Joe: Wow. I'm not even that clear on how we'd find the empty square in the first place...

You can move around a DOM tree using FAMILY relationships

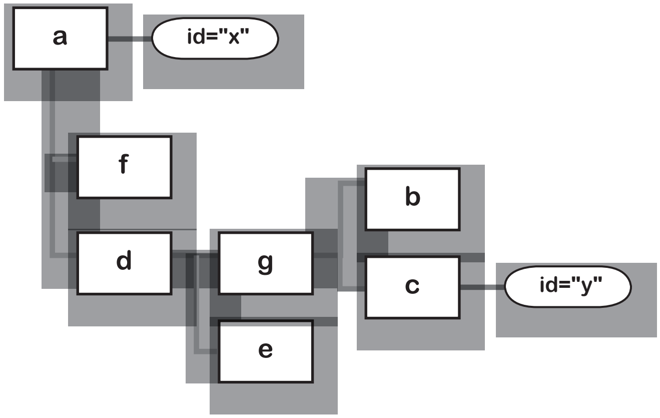
Suppose you wanted to find out the parent of an `<img>` or get a reference to the next `<td>` in a table. A DOM tree is all connected, and you can use the family-type properties of the DOM to move around in the tree.

`parentNode` moves up the tree, `childNodes` gives you an element's children, and you can move between nodes with `nextSibling` and `previousSibling`. You can also get an element's `firstChild` and `lastChild`. Take a look:



Sharpen your pencil

Below is a DOM tree and some JavaScript statements. In the JavaScript, each letter is a variable that represents the matching node in the DOM tree. Can you figure out which node each statement refers to?




```
document.getElementById("y");                 _______________

g.parent;                                     _______________

document.getElementById("y").nextSibling;     _______________

a.firstChild;                                 _______________

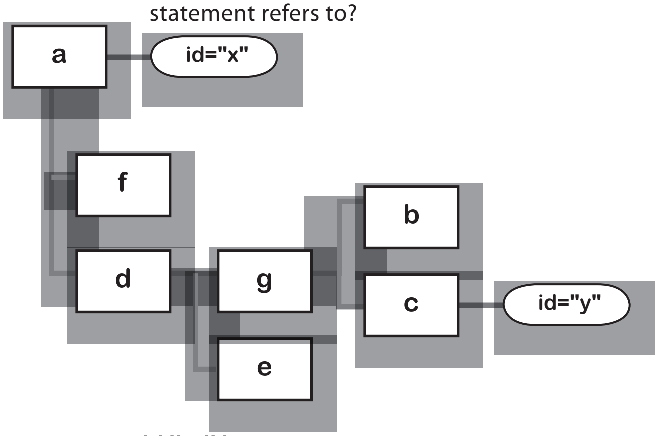
c.parent.parent;                              _______________

d.firstChild.lastChild;                       _______________

c.previousSibling;                            _______________
```

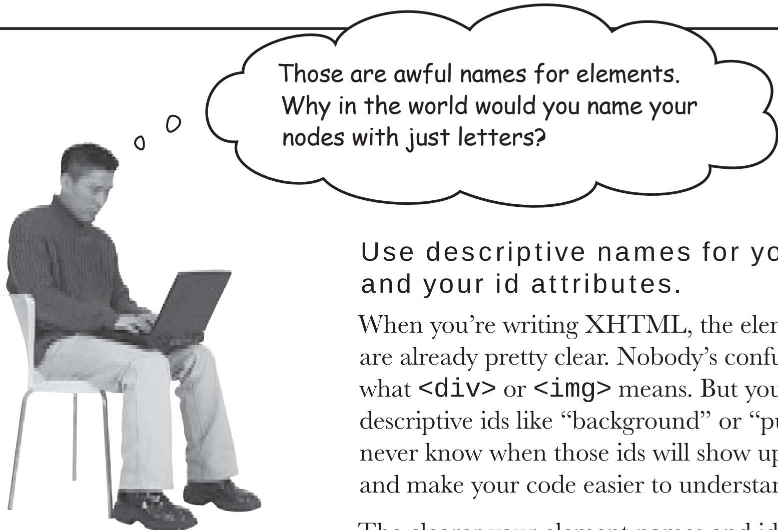
Sharpen your pencil Solution

Below is a DOM tree and some JavaScript statements. In the JavaScript, each letter is a variable that represents the matching node in the DOM tree. Can you figure out which node each statement refers to?



`document.getElementById("y");`	c
`g.parent;`	d
`document.getElementById("y").nextSibling;`	null
`a.firstChild;`	f
`c.parent.parent;`	d
`d.firstChild.lastChild;`	c
`c.previousSibling;`	b

This is a tricky one. Remember that the order of elements in the DOM tree reflects the order they're declared in the XHTML. Since there is no sibling below c, JavaScript returns a null.



Use descriptive names for your elements and your id attributes.

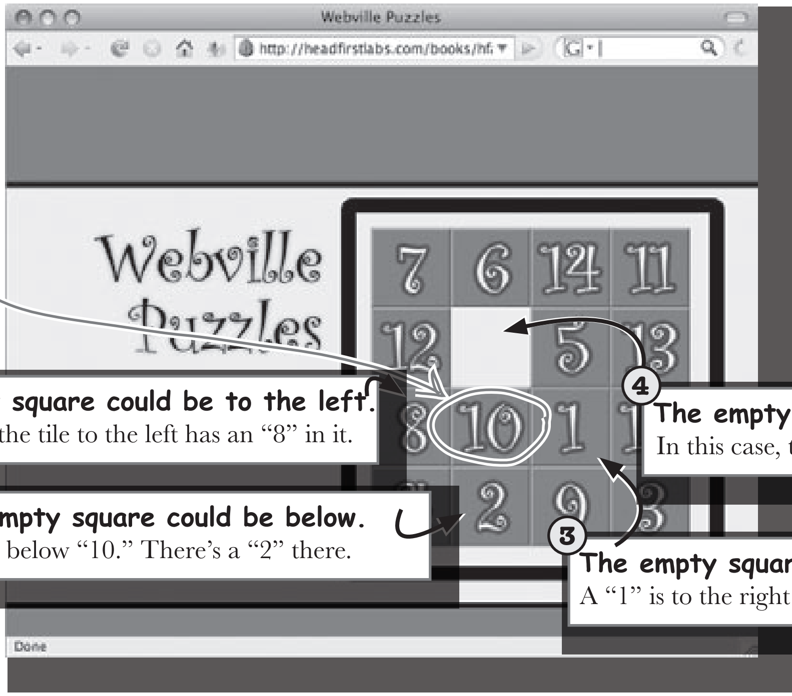
When you're writing XHTML, the element names are already pretty clear. Nobody's confused about what `<div>` or `<img>` means. But you should still use descriptive ids like "background" or "puzzleGrid." You never know when those ids will show up in your code, and make your code easier to understand... or harder.

The clearer your element names and ids, the clearer your code will be to you and other programmers.

Sharpen your pencil Solution

There were two more possibilities for where an empty tile could be on a puzzle grid. Did you figure out what they were?

The user clicked the "10 tile."



1 The empty square could be to the left.
In this case, the tile to the left has an "8" in it.

2 The empty square could be below.
It's not below "10." There's a "2" there.

3 The empty square could be to the right.
A "1" is to the right of "10."

4 The empty square could be above.
In this case, the empty square ***is*** above "10."

5 The empty square isn't next to the tile.
Only tiles next to an empty square can be swapped.

We've got to make sure a tile isn't surrounded by other non-empty tiles. If it is, it's locked and can't be moved.

6 The empty square was clicked on.
In this case, no tiles should be moved.

It's possible the tile clicked is the empty tile. That shouldn't cause a swap to occur, either.

Long Exercise

It's time to get busy building the rest of the Fifteen Puzzle Code. Here's your assignment:

1 Write a cellIsEmpty() function.

Given a node representing a `<td>`, figure out if the image in that cell is the empty image. To help you out, here's the XHTML for the empty cell:

```
<td id="cell22">
  <img src="images/empty.png" alt="empty" width="69" height="69" />
</td>
```

Here's part of the function to get you started:

```
function cellIsEmpty(cell) {
  var image = ......................................
  if (..........................................)
    return true;
  else
    return false;
}
```

cell is a node in the browser's DOM tree that represents a <td>.

2 Look for an empty cell in the tileClick() event handler.

Let's start building `tileClick()`, the event handler we attached to each table cell. First, we need to check for an empty cell. If the clicked-on cell was empty, let's show a message indicating the user needs to click on a different tile.

```
function tileClick() {
  if (cellIsEmpty(...........)) {
    alert("Please click on a numbered tile.");
    .................................
  }
}
```

3 Figure out what row and column was clicked.

Here's the XHTML for a couple of cells in the puzzle:

```
<td id="cell13">
 <img src="images/14.png" alt="14" width="69" height="69" />
</td>
<td id="cell14">
 <img src="images/11.png" alt="11" width="69" height="69" />
</td>
</tr>
<tr>
 <td id="cell21">
  <img src="images/12.png" alt="12" width="69" height="69" />
 </td>
 ... etc ...
```

Given this (and the rest of the XHTML, on page 246), can you figure out how to get the row and column of the clicked on tile?

```
var currentRow = this.................;
var currentCol = this.................;
```

Hint: you'll need to use JavaScript's charAt(int position) function at least once.

For the string "cows gone wild," charAt(2) returns "w."

4 Finish up the tileClick() event handler.

Once we've made sure that the empty tile wasn't clicked, and gotten the current row and column, you have everything you need. Your job is to handle the 5 remaining possible situations for where the empty square is, and then if possible, swap the selected tile with the empty square.

To get you started, here's the code to check above the selected tile:

```
// Check above
if (currentRow > 1) {
  var testRow = Number(currentRow) - 1;
  var testCellId = "cell" + testRow + currentCol;
  var testCell = document.getElementById(testCellId);
  if (cellIsEmpty(testCell)) {
    swapTiles(this, testCell);
    return;
  }
}
```

Only check above if we're not on row 1.

Number converts text into a numeric format.

JavaScript automatically turns these numbers into strings when they're added together with another string.

The id of the cell above is "cell," and then (currentRow –1), and then the current column number.

Get the test cell based on its id...

...see if it's the empty square...

...and then swap out the current cell and the empty square.

If we swapped tiles, we're done!

The rest of `tileClick()` is up to you. Refer back to the different possibilities you have to handle from page 265, and good luck!

Long Exercise Solution

Your job was to build the cellIsEmpty() function, and then complete the clickTile() event handler. Did you figure everything out? Here's what we did:

```
function cellIsEmpty(cell) {
  var image = cell.firstChild;
  if ( image.alt == "empty" )
    return true;
  else
    return false;
}
```

The first child of each table cell is its <img>.

The empty image has an alt tag of "empty."



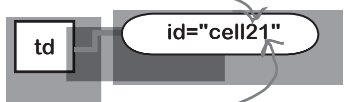

```
function tileClick() {
  if (cellIsEmpty( this )) {
    alert("Please click on a numbered tile.");
    return;
  }

  var currentRow = this. id .charAt(4) ;
  var currentCol = this. id .charAt(5) ;


  // Check above
  if (currentRow > 1) {
    var testRow = Number(currentRow) - 1;
    var testCellId = "cell" + testRow + currentCol;
    var testCell = document.getElementById(testCellId);
    if (cellIsEmpty(testCell)) {
      swapTiles(this, testCell);
      return;
    }
```

"this" in tileClick() is the activated object. That's the clicked-on tile.

Be sure to return if the empty tile was clicked on.



The id of each table cell gives the row, and then the column.

```
  }

  // Check below
  if (currentRow < 4) {
    var testRow = Number(currentRow) + 1;
    var testCellId = "cell" + testRow + currentCol;
    var testCell = document.getElementById(testCellId);
    if (cellIsEmpty(testCell)) {
      swapTiles(this, testCell);
      return;
    }
  }
  // Check to the left
  if (currentCol > 1) {
    var testCol = Number(currentCol) - 1;
    var testCellId = "cell" + currentRow + testCol;
    var testCell = document.getElementById(testCellId);
    if (cellIsEmpty(testCell)) {
      swapTiles(this, testCell);
      return;
    }
  }
  // Check to the right
  if (currentCol < 4) {
    var testCol = Number(currentCol) + 1;
    var testCellId = "cell" + currentRow + testCol;
    var testCell = document.getElementById(testCellId);
    if (cellIsEmpty(testCell)) {
      swapTiles(this, testCell);
      return;
    }
  }

  // The clicked-on cell is locked
  alert("Please click a tile next to an empty cell.");
}
```

Make sure that we're not on the bottom row.

Get the cell one row down, in the same column.

If the target cell is empty, do a swap.

Now we're looking side-to-side. Make sure we're not in the leftmost column.

Find the cell one column over, in the same row.

See if that cell is empty. If so, swap and return.

Each of these cases—below, left, and right—follow the same pattern.

If we got here, the clicked-on tile isn't empty, and it's not next to an empty square.

Let's give the user some feedback, so they know what to do.

there are no Dumb Questions

Q: **Wow, that was a lot of code. Am I supposed to understand all that?**

A: It is a lot of code, but if you walk through things step by step, it should all make sense to you. There's not a lot of new stuff in there, but there's definitely more DOM and positioning than you've done up to this point.

Q: **And all of this is DOM code?**

A: Well, there's really no such thing as DOM code. It's all JavaScript, and lots of it does use the DOM.

Q: **So which parts use the DOM?**

A: Anytime you use a property that moves around the tree, you're using the DOM in some form. So `firstChild` and `previousSibling` are DOM properties. But code that uses `getElementById()` is also using the DOM because that's a property on the `document` object. `document` is the top-level object of a browser's DOM tree.

The DOM is great for code that involves positioning and moving nodes around on a page.

Q: **Is it safe to assume the id of a table cell has the row and column in it?**

A: If you have control of the XHTML, like we do, it's safe. Since the Webville Puzzles company set up their XHTML so that table cells had those handy ids, we were able to figure out a cell's position by its id. If you had a different setup, you might need to figure out the cell's position relative to other cells and rows.

Q: **We could do that with the DOM, too, right?**

A: You bet. You'd probably be using some sort of counter, as well as `previousSibling` to figure out how many `<td>`'s over you are. And you'd need `parentNode` and similar properties to see which row you were on.

Q: **So this DOM stuff can get pretty complex, can't it?**

A: It can, very fast. Although lots of times, you'll only need to get as complex as we had to for the Fifteen Puzzle. In fact, with just the properties you've already learned, you're halfway to being a DOM master!

Q: **Halfway? What's the other half?**

A: So far, we've only moved around nodes in the DOM. In the next chapter, we'll look at creating new nodes and adding those to the tree on the fly.

Q: **Back to that code... so the firstChild of a table cell is always the image in that cell?**

A: That's the way `cellIsEmpty()` is written right now, yes. Can you think of a case where an image would *not* be the first child of a table cell?

Q: **If an image isn't the first child of a table cell, that screws things up, doesn't it?**

A: It sure does.

Q: **Well, didn't we do the same thing in swapTiles(), back on page 254? We assume the image is the firstChild there, too, right?**

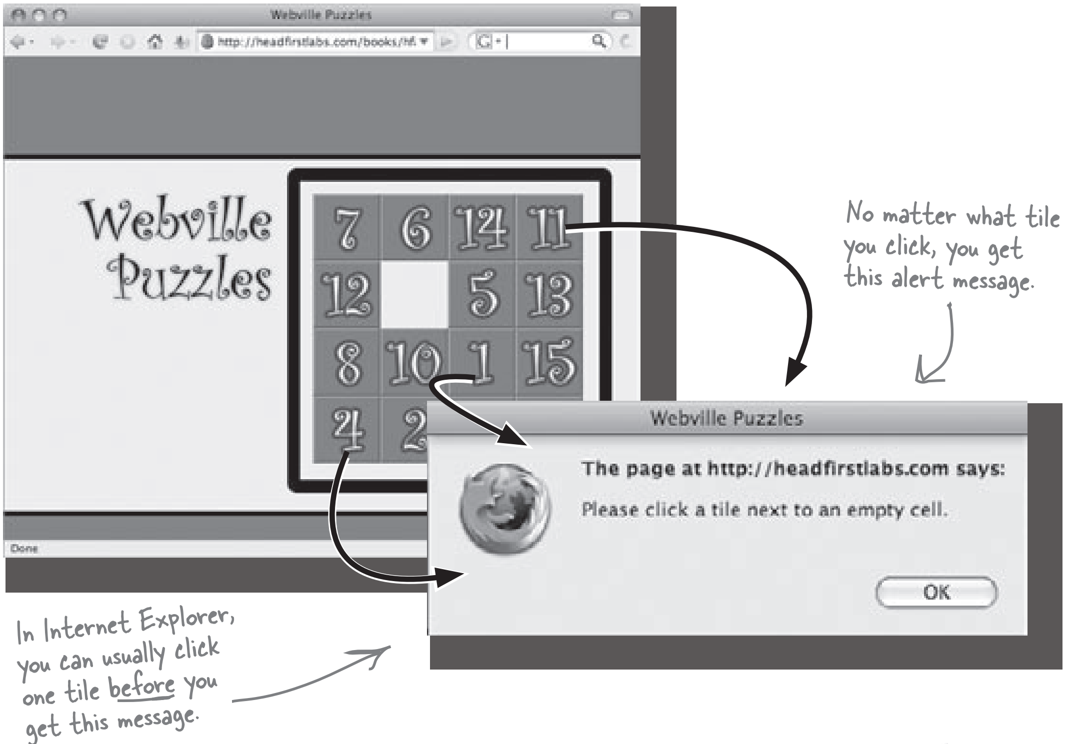
A: Exactly right. So would that assumption ever be false?

Q: **Who's asking the questions here, anyway?**

A: Maybe we should actually test out the fifteen puzzle, and see what happens.

Test Drive

Open your copy of fifteen.js, and add the code for cellIsEmpty() and tileClick(). Make sure you've got initPage() and swapTiles() working, too. Load things up. Does the puzzle work?



Exercise

Here's the XHTML for each table cell in the fifteen puzzle web page:

```
<td id="cell22">
  <img src="images/empty.png" alt="empty" width="69" height="69" />
</td>
```

Is there any difference between that XHTML and this fragment:

```
<td id="cell22"><img src="images/empty.png" alt="empty" width="69" height="69" /></td>
```

Take a close look at swapTiles() and cellIsEmpty(). Do you see a problem related to the difference in the two XHTML fragments shown above?

A DOM tree has nodes for EVERYTHING in your web page

Most XHTML pages don't have every element, from the opening `<html>` to the closing `</html>` crammed onto one line. That would be a real pain to read. Instead, your page is full of spaces, tabs, and returns (sometimes called "end-of-lines"):

You've got returns, or end-of-lines, to split up the page and make it easier to read.

You also have spaces or tabs for indentation.

```
<table cellspacing="0" cellpadding="0">↵
␣<tr>↵
␣␣<td id="cell11">↵
␣␣␣<img src="images/07.png" alt="7" width="69" height="69" />↵
␣␣</td>↵
␣␣<td id="cell12">↵
␣␣␣<img src="images/06.png" alt="6" width="69" height="69" />↵
␣␣</td>↵
␣␣<td id="cell13">↵
␣␣␣<img src="images/14.png" alt="14" width="69" height="69" />↵
␣␣</td>↵
␣␣<td id="cell14">↵
␣␣␣<img src="images/11.png" alt="11" width="69" height="69" />↵
␣␣</td>↵
␣</tr> ↵
  ... etc ...
</table>↵
```

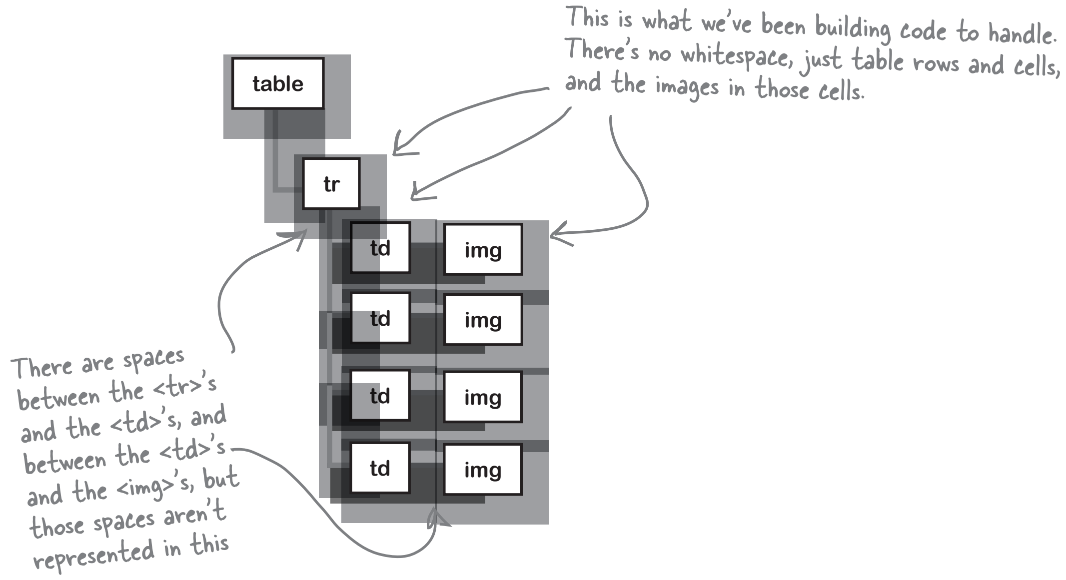
Those spaces are nodes, too

Even though those spaces are invisible to you, the browser tries to figure out what to do with them. Usually they get represented by text nodes in your DOM tree. So a `<table>` node might have lots of text nodes full of spaces in addition to all the `<tr>` children you'd expect.

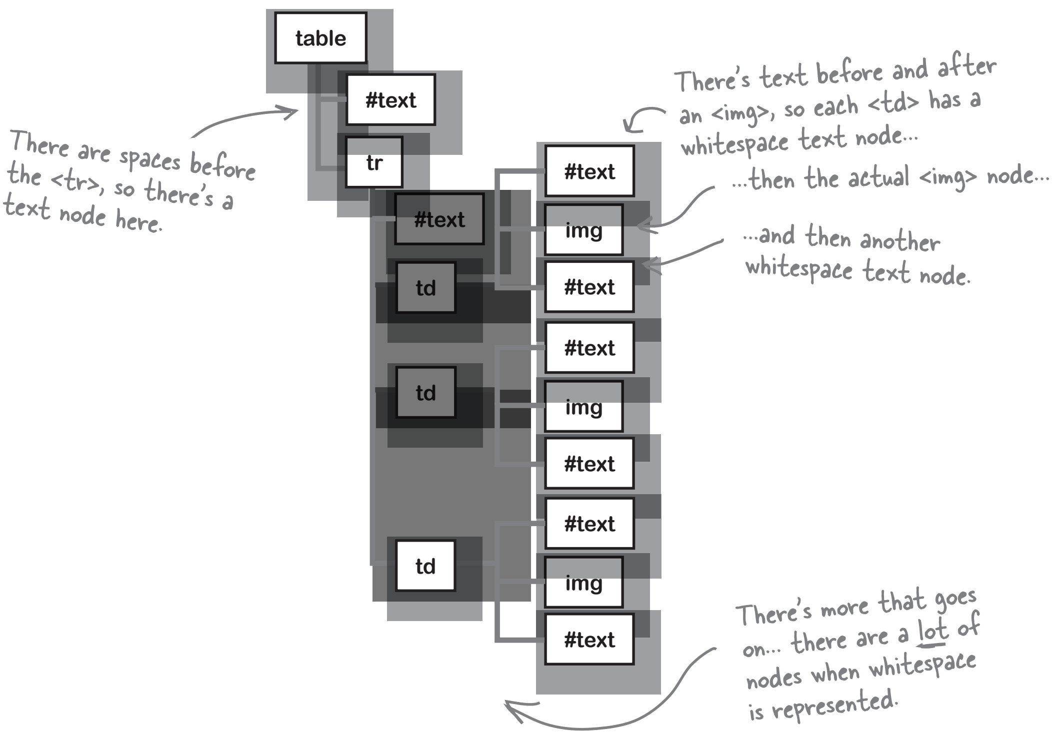
The bad news is that not all browsers do things the same way. So sometimes you get empty text nodes, and sometimes you don't. It's up to you to account for these text nodes, but you can't assume they'll always be there. Sounds a bit confusing, doesn't it?

There are some inconsistencies in how browsers treat whitespace. Never assume a browser will always ignore, or always represent, whitespace.

One browser might create a DOM tree for your page that looks like this:

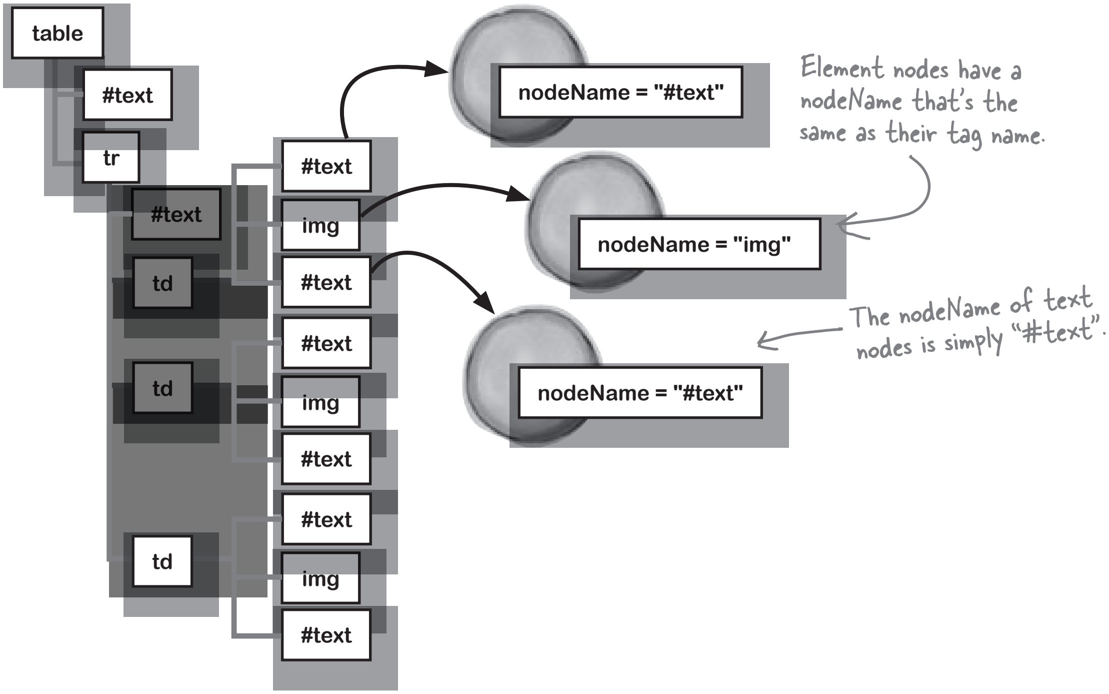


Another browser might create a different DOM tree for the same XHTML:



The nodeName of a text node is "#text"

A text node always has a `nodeName` property with a value of "#text." So you can find out if a node is a text node by checking its `nodeName`:

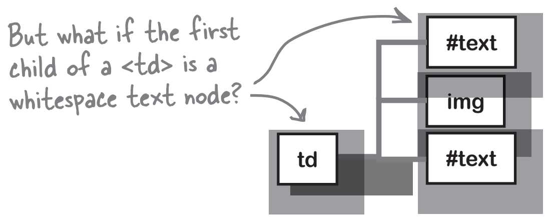


swapTiles() and cellIsEmpty() don't take whitespace nodes into account

The problem with our code is that our functions are assuming that the `<img>` in a table cell is the first child of a `<td>`:

```
function swapTiles(selectedCell, destinationCell) {
  selectedImage = selectedCell.firstChild;
  destinationImage = destinationCell.firstChild;
  selectedCell.appendChild(destinationImage);
  destinationCell.appendChild(selectedImage);
}
```

This will only work if the first child is an <img> element.



Sharpen your pencil

We've got to deal with browsers that are creating whitespace nodes in their DOM trees. See if you can fill in the blanks to fix up the swapTiles() and cellIsEmpty() functions below:

```
function swapTiles(selectedCell, destinationCell) {
  selectedImage = selectedCell.firstChild;
  while (selectedImage._______________ == _________) {
    selectedImage = selectedImage._______________;
  }
  destinationImage = destinationCell.firstChild;
  while (destinationImage._______________ == _________) {
    destinationImage = destinationImage._______________;
  }
  selectedCell.appendChild(destinationImage);
  destinationCell.appendChild(selectedImage);
}

function cellIsEmpty(cell) {
  var image = cell.firstChild;
  while (image.___________ == ____________) {
    image = image.______________;
  }
  if (image.alt == "empty")
    return true;
  else
    return false;
}
```

there are no Dumb Questions

Q: If the nodeName of a text node is always "#text", how can I get the text in that node?

A: Text nodes store the text they represent in a property called `nodeValue`. So the nodeValue for a whitespace node would be "" (an empty value), or possibly " " (two spaces).

Q: Shouldn't we be checking to see if text nodes have a nodeValue of whitespace, then?

A: In the table cells in the fifteen puzzle, there's no need to check the `nodeValue`. Since we only care about `<img>` nodes, we don't care about anything else. So we can skip over any node that's a text node.

Sharpen your pencil Solution

Were you able to figure out how to skip over the whitespace text nodes in cellIsEmpty() and swapTiles()?

```
function swapTiles(selectedCell, destinationCell) {
  selectedImage = selectedCell.firstChild;
  while (selectedImage.__nodeName__ == __"#text"__) {
    selectedImage = selectedImage.__nextSibling__;
  }
  destinationImage = destinationCell.firstChild;
  while (destinationImage.__nodeName__ == __"#text"__) {
    destinationImage = destinationImage.__nextSibling__;
  }
  selectedCell.appendChild(destinationImage);
  destinationCell.appendChild(selectedImage);
}

function cellIsEmpty(cell) {
  var image = cell.firstChild;
  while (image.__nodeName__ == __"#text"__) {
    image = image.__nextSibling__;
  }
  if (image.alt == "empty")
    return true;
  else
    return false;
}
```

We can find out if we've got a text node by comparing the nodeName to "#text".

If we've got a text node, move to the next sibling and try again.

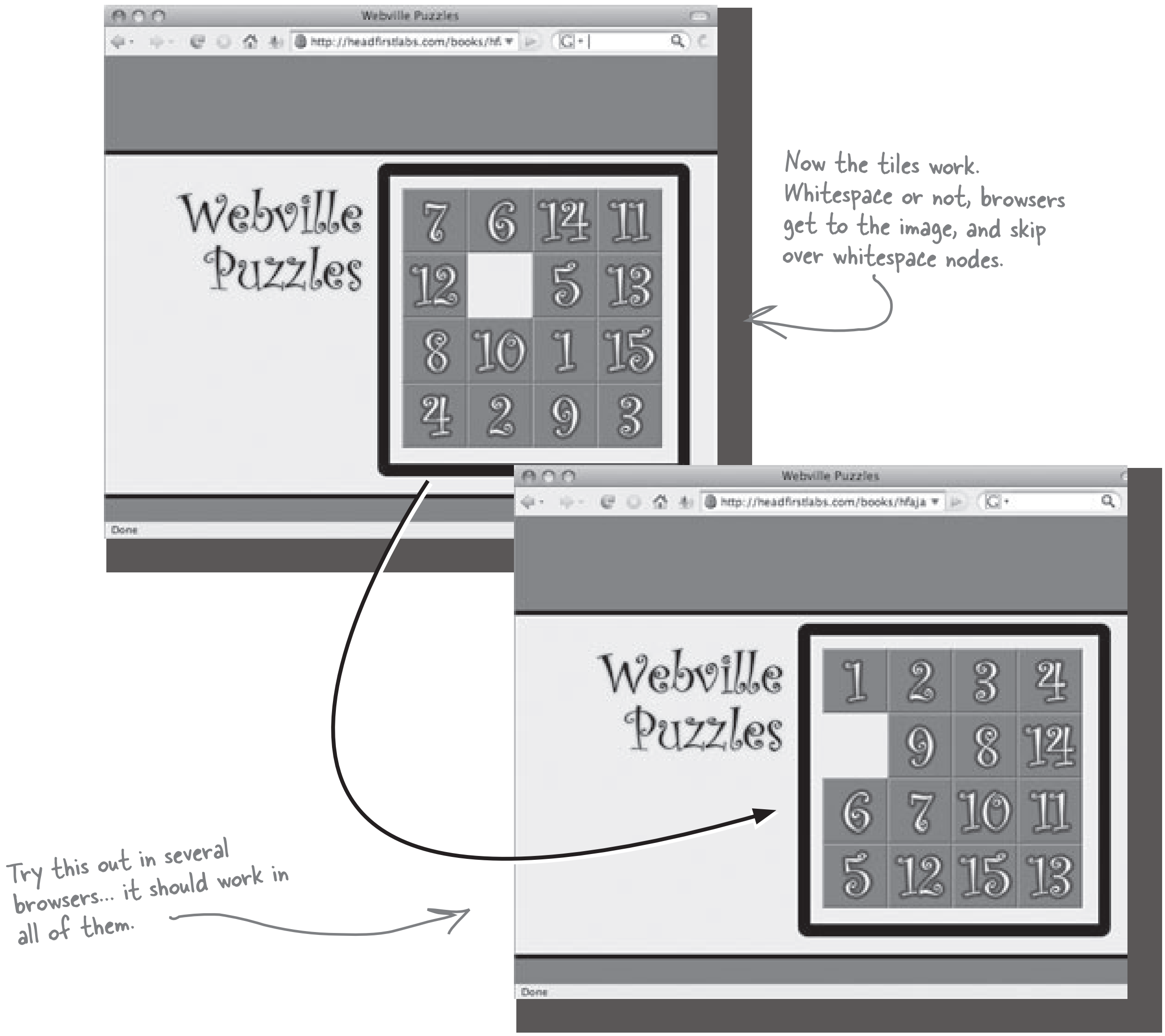
Make sure you've got open-quote, then the # symbol, then text, then close quote.

All three of these cases use the same basic pattern: as long as the current node is text, go to the next node.

Check for a text node, and use DOM methods to move to the next node if we've got text.

Test Drive

Update your versions of swapTiles() and cellIsEmpty(). Try the puzzle again... you should be able to move tiles around without a problem.



Did I win? Did I win?

All that's left is to figure out when a player's won. Then, every time two tiles are swapped, we can check this function to see if the board is in order. If it is, the player's solved the puzzle.

Here's a `puzzleIsComplete()` function that uses the names of each image to see if all the tiles are in order:

First, we get all the <img> tags in the grid.

```
function puzzleIsComplete() {

  var tiles = document.getElementById("puzzleGrid").getElementsByTagName("img");

  var tileOrder = "";

  for (var i=0; i<tiles.length; i++) {

    var num = tiles[i].src.substr(-6,2);

    if (num != "ty")

      tileOrder += num;

  }

  if (tileOrder == "010203040506070809101112131415")

    return true;

  return false;

}
```

We iterate over each tile image.

If you go back 6 characters from the end of the src of the image, you'll be at the image name: 02.png or empty.png.

We want just the numeric part, so that's 2 from –6 characters back.

We don't care about the empty image... we ignore it. Since the two characters returned by substr() for the empty image are "ty", check for that and ignore it.

If it's not the empty image, add the number (as a string) to our hash string.

If the numbers are in order, the puzzle's complete.

there are no Dumb Questions

Q: substr(-6, 2)? I don't get it.

A: A negative number means start at the end of the string, and count back. Since "02.png" is 6 characters, we want to go back from the *end* of the string by 6 characters.

Then, we want 2 characters of that substring, so we get "02" or "15." So you use `substr(-6,2)`.

Q: What in the world is that weird number you're comparing the hash string to?

A: It's just every number in the puzzle, in order: 01, then 02, then 03, and so on, all the way to 15. Since the hash represents the orders of the tiles, we're comparing them to a string that represents the tiles in order.

Q: But what about the empty tile?

A: It doesn't matter where the empty tile is as long as the numbers are in order. That's why we drop the empty tile from being part of the hash string.

But seriously... did I win?

There's even a special class that Webville Puzzles put in their CSS for showing a winning animation. The class is called "win," and when the puzzle is solved, you can set the `<div>` with an `id` of "puzzleGrid" to use this class and display the animation.

That means we just need to check if the puzzle is solved every time we swap tiles.

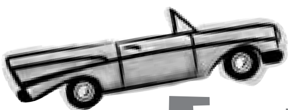
```
function swapTiles(selectedCell, destinationCell) {
  selectedImage = selectedCell.firstChild;
  while (selectedImage.nodeName == "#text") {
    selectedImage = selectedImage.nextSibling;
  }
  destinationImage = destinationCell.firstChild;
  while (destinationImage.nodeName == "#text") {
    destinationImage = destinationImage.nextSibling;
  }

  selectedCell.appendChild(destinationImage);
  destinationCell.appendChild(selectedImage);

  if (puzzleIsComplete()) {
    document.getElementById(“puzzleGrid”).className = “win”;
  }
}
```

Every time we swap tiles, we need to see if the new arrangement makes the puzzle complete.

If the puzzle's solved, change the CSS class for the puzzleGrid <div>.



Test Drive

You need to add the puzzleIsComplete() function to your JavaScript, and update swapTiles(). Then, try everything out. But you've got to solve the puzzle to see the winning animation...

...good luck!



The DOM is just a tool, and you won't use it all the time... or sometimes, all that much.

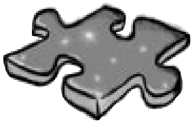
You'll rarely write an application that is mostly DOM-related code. But when you're writing your JavaScript, and you really need that *next* table cell, or the containing element of an image, then the DOM is the perfect tool.

And, even more importantly, without the DOM, there's really no way to get around a page, especially if every element on your page doesn't have an `id` attribute. The DOM is just one more tool you can use to take control of your web pages.

In the next chapter, you're going to see how the DOM lets you do more than just move things around... it lets you create elements and text on the fly, and put them anywhere on the page you want.

The DOM is a great tool for getting around within a web page.

It also makes it easy to find elements that DON'T have an id attribute.



DOMAcrostic

Take some time to sit back and give your right brain something to do. Answer the questions up top, and then use the answer letters to fill in the secret message.

This method returns a specific element based on its ID

1 2 3 4 5 6 7 8 9 10 12 13 14 15

This property returns all the children of an element

16 17 18 19 20 21 22 23 24 25

The browser translates this into an element tree

26 27 28 29 30 31

This element property represents the element's container

32 33 34 35 36 37

This is what the browser creates for you

38 39 40   41 42 43 44

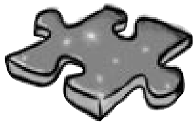
This element gives you access to the whole tree

45 46 47 48 49 50 51 52

49 27 25 41 35 28   52 17 8   45 22 7   33 51 20

13 39 48   26 33 25 10 2 34

13 46 30 34   32 27 1 43



DOMAcrostic

Your job was to answer the questions up top, and then use the answer letters to fill in the secret message.

This method returns a specific element based on its ID

G	E	T	E	L	E	M	E	N	T	B	Y	I	D
1	2	3	4	5	6	7	8	9	10	12	13	14	15

This property returns all the children of an element

C	H	I	L	D	N	O	D	E	S
16	17	18	19	20	21	22	23	24	25

This property is an array of nodes.

The browser translates this into an element tree

M	A	R	K	U	P
26	27	28	29	30	31

This element property represents the element's container

P	A	R	E	N	T
32	33	34	35	36	37

A node that has children "contains" those children.

This is what the browser creates for you

D	O	M		T	R	E	E
38	39	40		41	42	43	44

This element gives you access to the whole tree

D	O	C	U	M	E	N	T
45	46	47	48	49	50	51	52

The document object contains everything else in the DOM tree.

M	A	S	T	E	R		T	H	E		D	O	M		A	N	D
49	27	25	41	35	28		52	17	8		45	22	7		33	51	20

Y	O	U		M	A	S	T	E	R
13	39	48		26	33	25	10	2	34

Y	O	U	R		P	A	G	E
13	46	30	34		32	27	1	43

7 manipulating the DOM

My wish is your command



Sometimes you just need a little ~~mind~~ DOM control.

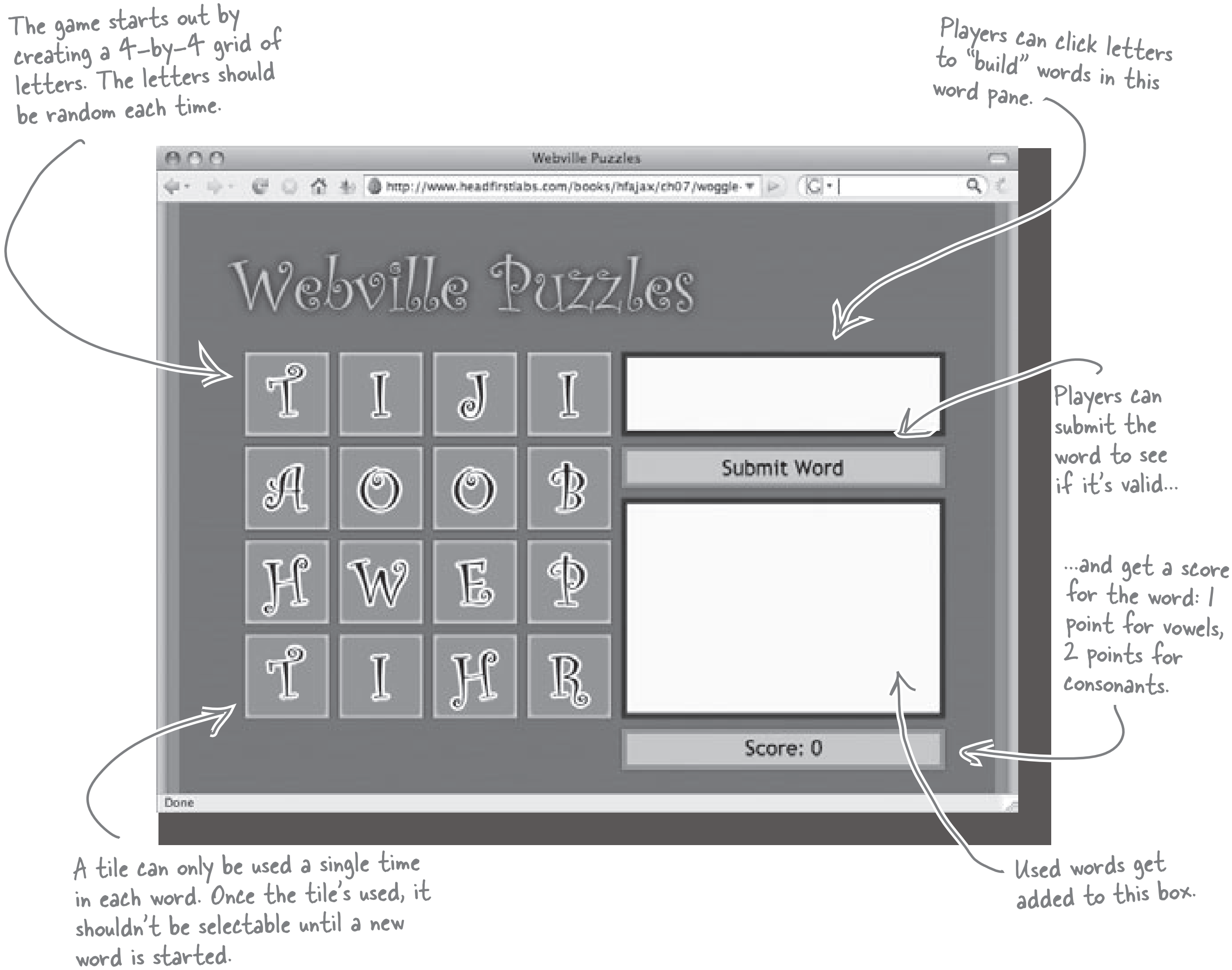
It's great to know that web browsers turn your XHTML into DOM trees. And you can do a lot by moving around within those trees. But real power is **taking control of a DOM tree** and making the tree look like *you* want it to. Sometimes what you really need is to **add a new element** and some text, or to **remove an element**, like an <img>, from a page altogether. You can do all of that and more with the DOM, and along the way, **banish that troublesome innerHTML property** altogether. The result? Code that can do more to a page, *without* having to mix presentation and structure in with your JavaScript.

Webville Puzzles... the franchise

All the cool kids have been playing the Fifteen Puzzle you developed for Webville Puzzles. The company's been making so much on subscription fees that they want a new puzzle... and they've come to you to build the interactivity.

This time, the company wants something a little more educational: **Woggle**, an online word generation game. They've already built the XHTML, and even know exactly how they want the puzzle to work.

Here's the initial Woggle page:



Exercise

There's a lot to build to get Woggle working, and of course the company wants their new app working immediately. Before you dig into the XHTML and CSS, think about what tasks are involved, and what JavaScript each one will need. Try and list each basic task for which you'll need to write code, and then make notes about what tools and techniques you might use for that task.

Task 1: ....

Notes: ....

....

....

....

Task 2: ....

Notes: ....

....

....

....

Task 3: ....

Notes: ....

....

....

....

Task 4: ....

Notes: ....

....

....

....

Task 5: ....

Notes: ....

....

....

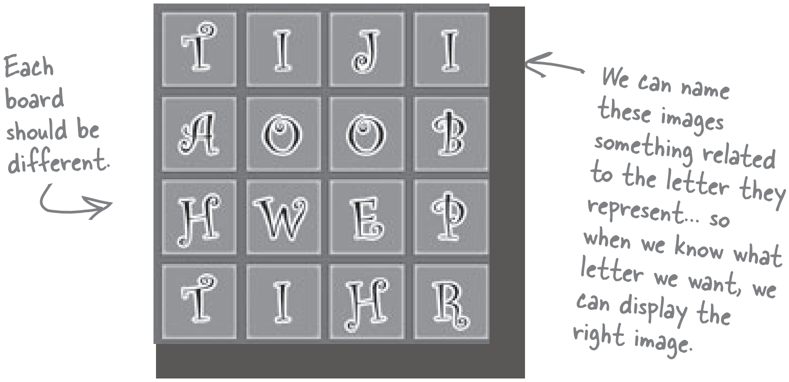
....

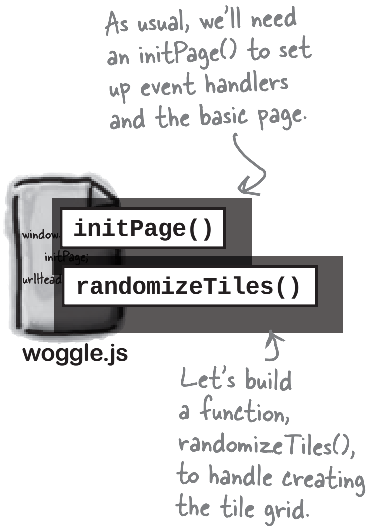
Exercise Solution

Your job was to figure out the basic tasks we'd need to take care of to get Woggle working. Here's what we came up with. You might have some differences in your details, but make sure you got these same core ideas down in some form or fashion.

Task 1: Set up the game board with random tiles

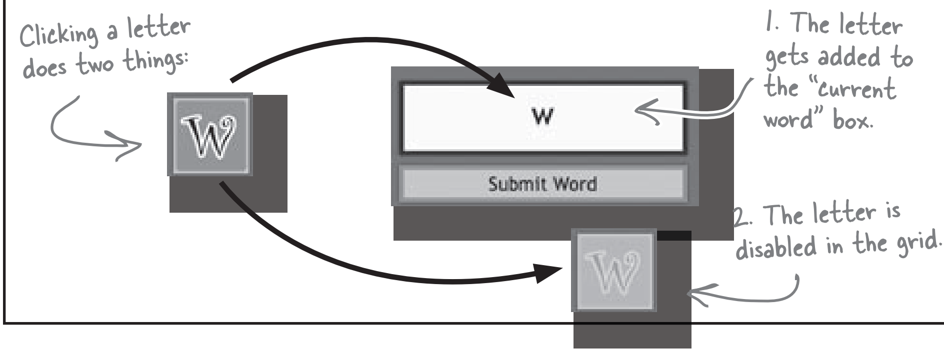
Notes: We need a way to come up with a random set of letters. Then we've got to display the right image for each letter on the 4x4 game board. This probably should all be done in an initPage() type of function.

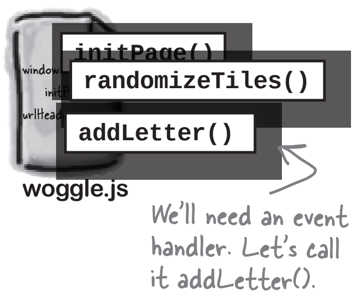




Task 2: Clicking on a tile adds the letter to the current word.

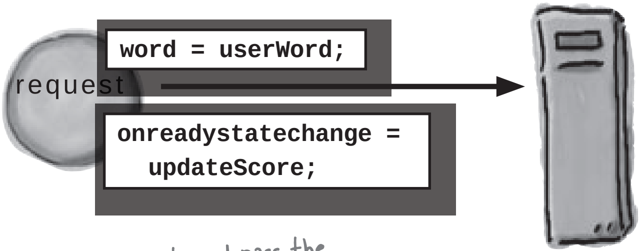
Notes: We need an event handler on each tile. The handler should figure out what letter was clicked, and add it to the "current word" box over on the right. Then the tile that was clicked should be disabled in the grid.

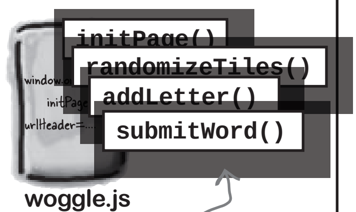




Task 3: **Users can submit words to the server.**

Notes: **When a user clicks "Submit Word," the current word is sent to the server-side program. We'll also need to register a callback to deal with the server's response.**



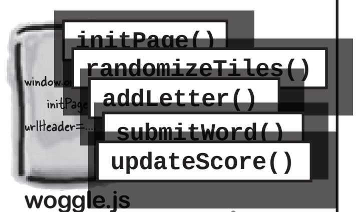


We can create a request, and pass the server the current word and a callback to run when the server responds.

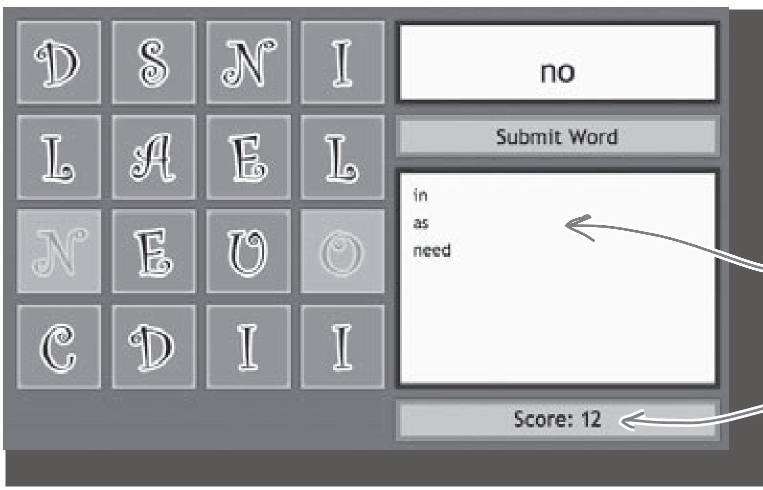
submitWord() can set up and send a server-side program request.

Task 4: **Update the score using the server's response.**

Notes: **When the server responds, we've got to update the score, and add a valid word to the "used words" box. We've also got to remove the word from the "current word" box and enable the tiles again.**



One more function... our callback.



Each time a word's accepted, all the tiles are enabled for a new word.

Valid words get added to this box...

...and the score gets updated with each valid word.

Woggle doesn't use table cells for the tiles

Now that we've got a plan, let's look at the XHTML for the Woggle game. The designers at Woggle have heard some bad things about tables, so the page is structured a bit differently. Each tile is represented by an <a> element this time, instead of being in a table cell. Better for them, but that might mean a little more work for us.

Here's what the XHTML looks like:

```
<html>
<head>
 <title>Webville Puzzles</title>
 <link rel="stylesheet" href="css/puzzle.css" type="text/css" />
 <script src="scripts/utils.js" type="text/javascript"></script>
 <script src="scripts/woggle.js" type="text/javascript"></script>
 </head>
<body>
 <div id="background">
  <h1 id="logotype">Webville Puzzles</h1>
  <div id="letterbox">
   <a href="#" class="tile t11"></a>
   <a href="#" class="tile t12"></a>
   <a href="#" class="tile t13"></a>
   <a href="#" class="tile t14"></a>
   <a href="#" class="tile t21"></a>
   <a href="#" class="tile t22"></a>
   <a href="#" class="tile t23"></a>
   <a href="#" class="tile t24"></a>
   <a href="#" class="tile t31”></a>
   <a href="#" class="tile t32"></a>
   <a href="#" class="tile t33"></a>
   <a href="#" class="tile t34"></a>
   <a href="#" class="tile t41"></a>
   <a href="#" class="tile t42"></a>
   <a href="#" class="tile t43"></a>
   <a href="#" class="tile t44"></a>
  </div>
  <div id="currentWord"></div>
   <div id="submit"><a href="#">Submit Word</a></div>
   <div id="wordListBg">
    <div id="wordList"></div>
   </div>
  <div id="score">Score: 0</div>
 </div>
</body>
</html>
```

You should go ahead and add these script references. We'll use utils.js for creating a request object...

...and woggle.js is the script we'll build for our puzzle-specific functions.

Each set of 4 <a>'s represents one row in the puzzle grid.

Here's where the current word will go...

...and the "Submit Word" button will go here.

There's a place for words already used...

...and finally, the score.



woggle-puzzle.html

The tiles in the XHTML are CSS-positioned

Instead of putting the tiles inside of a table, each <a> element that represents a tile is given a general class ("tile") and then a specific class, indicating where on the board it is (for example, "t21"):

```
<a href="#" class="tile t21"></a>
```

"tile" is a general CSS class that applies to all tiles in the grid.

This is for the specific tile. 2 is the row, and 1 is the column. So this is the first column in the second row.

The CSS then uses both the general "tile" class and the specific tile class ("t21", "t42", etc.) to style and position the tiles.

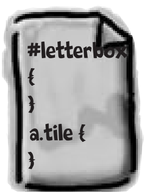
This CSS class applies to all tiles, so each element with a "tile" class gets these properties.

```
/* tile defaults */
#letterbox a.tile {
  background: url('../images/tiles.png') 120px 80px no-repeat;
  height: 80px;
  position: absolute;
  width: 80px;
}

/* tile positioning */
#letterbox a.t11 { top: 3px; left: 3px; }
#letterbox a.t12 { top: 3px; left: 93px; }
#letterbox a.t13 { top: 3px; left: 183px; }
#letterbox a.t14 { top: 3px; left: 273px; }
... etc ...
```

There's an entry in the CSS for each tile... 16 in all.

This CSS sets the position for each individual <a> that represents a tile.



puzzle.css



Download the XHTML and CSS for Woggle.

Visit **www.headfirstlabs.com**, and find the **chapter07** folder. You'll see the XHTML and CSS for Woggle. You should add the `<script>` tags to `woggle-puzzle.html`, and get ready to dig into some code.

there are no Dumb Questions

Q: CSS-positioned? I'm not sure I know what that means.

A: CSS-positioned just means that instead of relying on the structure of your XHTML to position something on a page, CSS is used instead. So if you want to CSS-position an `<a>` element, you give that element a class or id, and then in your CSS, set its `left`, `right`, `top`, and/or `bottom` properties, or use the `position` and `float` CSS attributes.

Q: Is that better than using tables?

A: A lot of people think so, especially web designers. By using CSS, you're relying on your CSS to handle presentation and positioning, rather than the way cells in a table line up. That's a more flexible approach to getting your page to look like you want.

Q: So which should I use? Tables or CSS-positioning?

A: Well, you really can't go wrong with CSS-positioning, because it's the easiest approach to getting things to look the same across browsers.

But more importantly, you should be able to write code that works with tables *or* CSS positioning. You can't always control the pages you write code for, so you need to be able to work with lots of different structures and types of pages.

Q: I don't understand how the CSS positioning actually worked, though. Can you explain that again?

A: Sure. Each tile is represented by an `<a>` in the XHTML. And each `<a>` has a `class` attribute, and actually has two classes: the general class, "tile," and a specific class representing that tile, like "t32." So the class for the tile on the third row, second column would be "tile t32."

Then, in the CSS, there are *two* selectors applied to each tile: the general rule, "tile," and the specific selector for a tile, like "t32." So you have selectors like `a.tile`, and `a.t32`. Both of those selectors get applied to a tile with a class of "tile t32."

The general rule handles common properties for all tiles, like height and width and look. The specific selector handles that tile's position on the page.

Q: Why are <a>'s used for the tiles? They're not links, right?

A: No, not really. That's just what Webville Puzzles used (maybe they've been checking out the tabs on Marcy's yoga site). It really doesn't matter what you use, as long as there's one element per tile, and you can position that element in the CSS.

There are a few considerations that using an `<a>` brings up for our event handlers, though, and we'll look at those a bit later.

Q: I don't see a button for "Submit Word." There's just a <div>. What gives?

A: You don't have to have an actual form button to make something *look* like a button. In this case, the Webville Puzzle designers are using a `<div>` with a background image that looks like a button for the "Submit Word" button. As long as we attach an event handler to that `<div>` to capture clicks, we can treat it like a button in our code, too.

You should be able to write code to work with ALL TYPES of pages... even if the structure of those pages isn't how YOU would have done things.

BRAIN POWER

An event handler attached to an <a> element usually returns either true or false. What do you think the browser uses that return value for?

Here are the tasks from page 286. First, we need to set up the board.

"We don't want TOTALLY random letters..."

The guys in the puzzle labs at Webville Puzzles just called. They've decided they don't want totally random letters for the board after all. Instead, they want letters to appear according to a letter frequency chart they're faxing over... that way, common letters like "e" and "t" show up in the grid a lot more often than uncommon ones like "z" and "w."

Here's what the guys faxed over to you.

Letter	# of times appears out of 100
a	8
b	1
c	3
d	3
e	12
f	2
g	2
h	6
i	7
j	1
k	1
l	4
m	2
n	6
o	8
p	2
q	6
r	6
s	8
t	3
u	2
v	2
w	1
x	1
y	2
z	1

Given 100 random letters from actual English words, "e" appears about 12 times.

Thanks,
Webville Puzzles

Sharpen your pencil

Let's get started. First, you need to build an initPage() and randomizeTiles() function. Here's what you know:

1. There's a class for each lettered tile in puzzle.css. The class for tile "a," for example, is called "la" (the letter "l" for letter, plus the letter the tile represents).

2. Webville Puzzles has faxed you a letter frequency table. There are 100 entries, with each letter represented the number of times out of 100 it typically appears. You need to represent that table as an array in your JavaScript. There should be 100 entries, where each entry is a single letter.

3. Randomly choosing a letter from the frequency table is like choosing a letter based on the frequency in which it appears in a word.

4. Math.floor(Math.random()*2000) will return a random number between 0 and 1999.

5. You'll need to use getElementById() and getElementsByTagName() each *at least* once.

Try to complete both initPage() and randomizeTiles() *before* you turn the page. Good luck!

Sharpen your pencil Solution

Your job was to use the information on page 291 to write code for initPage() and randomizeTiles(). You also may have come up with some other JavaScript outside of those functions... how did you do?

```
window.onload = initPage;
```

Did you remember this line? We've got to call initPage() to get anything working.

We made this a global variable. Any function in our JavaScript can use this table now.

```
var frequencyTable = new Array(
  "a", "a", "a", "a", "a", "a", "a", "a", "b", "c", "c", "c", "d", "d", "d",
  "e", "e", "e", "e", "e", "e", "e", "e", "e", "e", "e", "e", "f", "f", "g",
  "g", "h", "h", "h", "h", "h", "h", "i", "i", "i", "i", "i", "i", "i", "j",
  "k", "l", "l", "l", "l", "m", "m", "n", "n", "n", "n", "n", "n", "o", "o",
  "o", "o", "o", "o", "o", "o", "p", "p", "q", "q", "q", "q", "q", "q", "r",
  "r", "r", "r", "r", "r", "s", "s", "s", "s", "s", "s", "s", "s", "t", "t",
  "t", "u", "u", "v", "v", "w", "x", "y", "z");
```

Here's how we represented the letter frequency table. Each letter appears in the array the number of times out of 100 it shows up in the frequency table Webville Puzzles faxed us.

Put this JavaScript into a new file, woggle.js.



woggle.js

```
function initPage() {
  randomizeTiles();
}
```

All initPage() does right now is call randomizeTiles() to set up the puzzle grid.

```
function randomizeTiles() {

  var tiles = document.getElementById("letterbox").getElementsByTagName("a");

  for (i = 0; i < tiles.length; i++) {

    var index = Math.floor(Math.random() * 100);

    var letter = frequencyTable[index];

    tiles[i].className = tiles[i].className + ' l' + letter;
  }
}
```

First, we grab all the <a> elements in the letterbox <div>.

For each tile, we get a random index between 0 and 99...

...and choose a letter from the letter frequency table.

Next, we change the class name of the tile. To do this, keep the existing class name...

We separate each class name with a space.

...and then add "l" plus the letter chosen, like "la" for letter a, or "lw" for letter w.

Our presentation is ALL in our CSS

By using class names instead of directly inserting `<img>`'s into the XHTML, we've kept our JavaScript behavior totally separate from the presentation, content, and structure of the page. So suppose that for the tile in the second row, first column, the random index returned by `Math.floor(Math.random() * 100)` was 4.

Array indices are zero-based, so index "4" points to the 5th item (a, a, a, a, a).
1 2 3 4
5th item

The fifth entry in frequencyTable is "a", so that tile should be an "a." But instead of having our code insert the "a" image directly, and deal with image URLs, it just adds to the class of that tile:

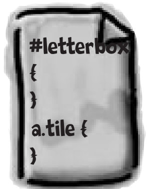
```
<a href="#" class="tile t21 la"></a>
```

This part was already in the page's XHTML for the tile.

This part gets added by randomizeTiles().

Now we use the CSS to say how that letter is displayed:

```
/* tile letters */
#letterbox a.la { background-position: 0px 0px; }
#letterbox a.lb { background-position: -80px 0px; }
#letterbox a.lc { background-position: -160px 0px; }
#letterbox a.ld { background-position: -240px 0px; }
#letterbox a.le { background-position: -320px 0px; }
... etc ...
```



puzzle.css

Now the designers have options

Since all the presentation is in the CSS, the designers of the page can do whatever they want to show the tiles. They might use a different background image for each letter. In the case of Woggle, though, the designers have used a single image for all tiles, `tiles.png`. `tiles.png,` It actually has every lettered tile in it, each in just the right size. That image is set as the background image in the selector for the `a.tile` class.

This is how we handled the "In Progress" and "Denied" image back in Chapter 5.

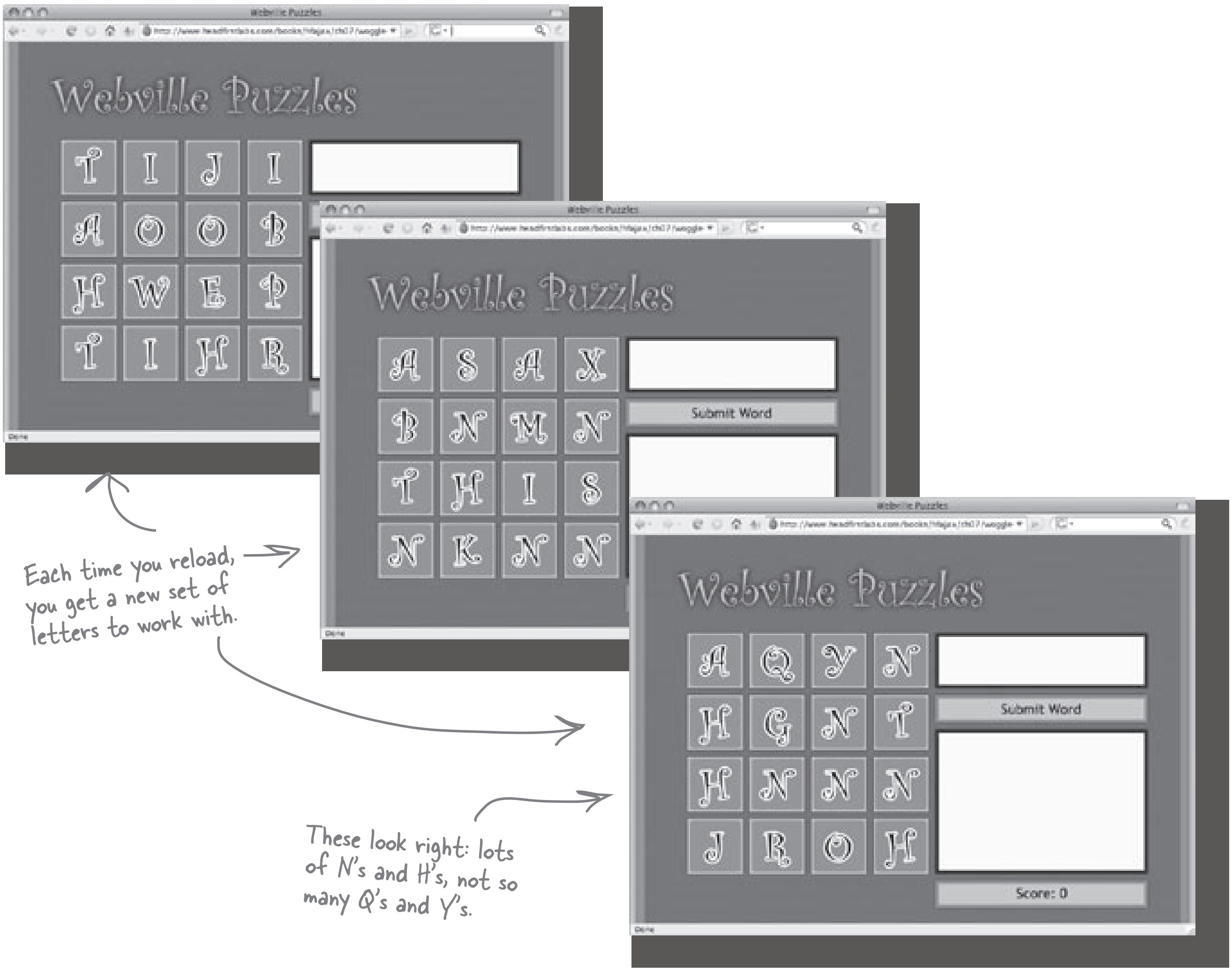
You can check out the CSS selector for a.tile back on page 289.

Then, in each letter-specific class, like `a.la` or `a.lw`, they've adjusted the position of the image so the right portion of that single image displays. Depending on the position, you get a different letter. And the designers can change the CSS anytime they want a new look... ***all without touching your code***.

Test Drive

Try out your early version of Woggle.

Download the samples files, and make sure you've added a reference to `woggle.js` in your version of `woggle-puzzle.html`. Then, load the Woggle main page in your browser... and load it again... and again...



We need a new event handler for handling tile clicks

Next up, we need to assign an event handler to the tiles on the grid. The handler needs to do several things:

1. **Figure out which letter was clicked.**
   All our handler will know about is the `<a>` element on the page that was clicked. From that, we've got to figure out which letter is shown on the tile that the clicked-upon `<a>` represents.

2. **Add a letter to the current word box.**
   Once we know what letter was selected, we've got to add that letter to the current word box in the "currentWord" `<div>`.

3. **Disable the clicked-on letter.**
   We've also got to keep the tile that was clicked from being clicked again. So we need to disable the letter. There's a CSS class for that, called "disabled," that works with the "tile," "t23," and "lw" classes already on each title.

This class handles formatting for all tiles.

This class represents the position of the tile.

This class represents the letter that's shown.

Sharpen your pencil

There are actually a couple of things missing from the list above... can you figure out what they are?

....

....

....

....

....

Start building the event handler for each tile click

The best way to write a big chunk of code is to take things one step at a time. First, let's build a skeleton for our handler. That's just a function block with a name, so we can hook everything up and test out code as we go.

Add this function to `woggle.js`:

We'll fill in each piece of this function as we go.

```
function addLetter() {
  // Figure out which letter was clicked

  // Add a letter to the current word box

  // Disable the clicked-on letter
}
```

woggle.js

The best way to write a large piece of code is to take things one step at a time.

We can assign an event handler in our randomizeTiles() function

Now let's go ahead and hook up our handler to each tile. We're already iterating over all the tiles in `randomizeTiles()`, so that seems like a good place to assign our event handler:

```
function randomizeTiles() {
  var tiles = document.getElementById("letterbox")
                      .getElementsByTagName("a");
  for (i = 0; i < tiles.length; i++) {
    var index = Math.floor(Math.random() * 100);
    var letter = frequencyTable[index];
    tiles[i].className = tiles[i].className +
                              ' l' + letter;
    tiles[i].onclick = addLetter;
  }
}
```

Now we can start testing our event handler as we write it.

woggle.js

Get each piece of code working BEFORE moving on to the next piece of code.

Property values are just strings in JavaScript

So far, we've mostly used the `className` property of an object to change a CSS class. For Woggle, we actually added classes to that property... but what if we want to read that value? Suppose the second tile on the third row represents the letter "b." That tile would have a `className` value that looks like this:

```
<a href="#" class="tile t32 lb"></a>
```

Third row — Second column — Letter "b"

So we've got a `className` property that has the letter of the tile that's represented... how can we get to that letter? Fortunately, JavaScript has a lot of useful string-handling utility functions:

substring

substring(startIndex, endIndex) returns a string from startIndex to endIndex, based on an existing string value.

```
var foo = "foolish".substring(0,3);
var is = "foolish".substring(4,6);
```

foo has the value "foo."

is has the value "is."

split

split(splitChar) splits a string into pieces separated by splitChar. The pieces are returned in an array.

```
var pieces = "Decide,Commit,Succeed".split(",");
alert(pieces[2] + "," + pieces[1] + "," + pieces[0]);
```

This will output "Succeed,Commit,Decide."

Sharpen your pencil

What code would you write to get the letter represented by the clicked-on tile?

....

....

....

....

Sharpen your pencil Solution

What code would you write to get the letter represented by the clicked-on tile?

```
<a href="#" class="tile t32 lb"></a>
```

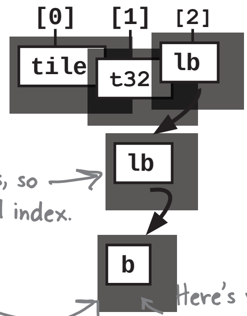
Here's a typical tile element.

```
var tileClasses = this.className.split(" ");
var letterClass = tileClasses[2];
var tileLetter = letterClass.substring(1, 2);
```

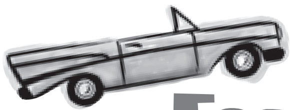
Split the classes from each other using the space character.

The letter class is the third class, so that's index 2 using a zero-based index.

We want a single character (length of 1), starting at index 2 in letterClass. So that's letterClass.substring(1, 2).



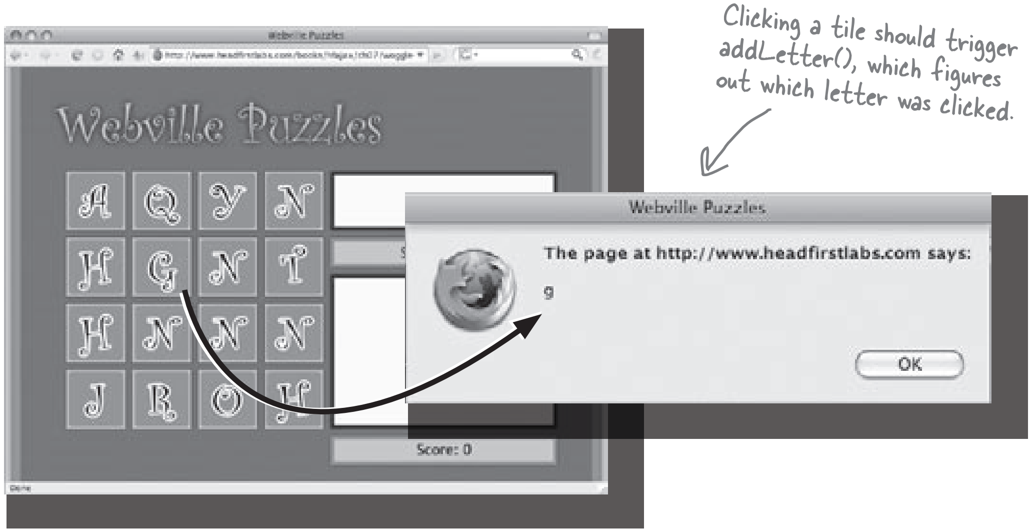
Here's what we want!



Test Drive

Test out your letter recognition.

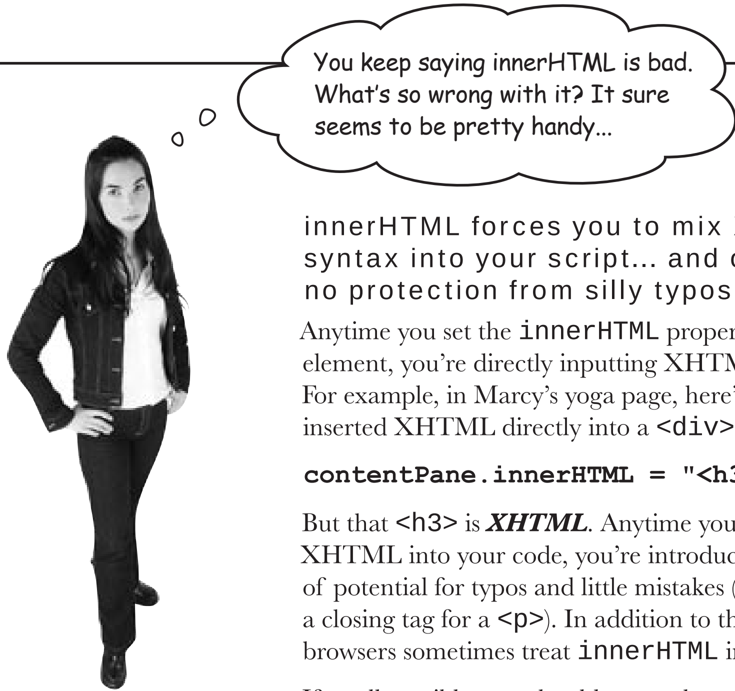
Add the code above to your `addLetter()` function, and also add an `alert()` statement to display the value of `tileLetter` at the end of the function. Then reload Woggle and see if everything's working...



Clicking a tile should trigger addLetter(), which figures out which letter was clicked.

Exercise

Now you've got to take the clicked-on letter, and add it into the currentWord <div>. How would you do that? Oh, and by the way... ***you can't use the innerHTML property on this one!***



innerHTML forces you to mix XHTML syntax into your script... and offers you no protection from silly typos, either.

Anytime you set the `innerHTML` property on an element, you're directly inputting XHTML into a page. For example, in Marcy's yoga page, here's where we inserted XHTML directly into a `<div>` with JavaScript:

```
contentPane.innerHTML = "<h3>" + hintText + "</h3>";
```

But that `<h3>` is ***XHTML***. Anytime you directly type XHTML into your code, you're introducing all sorts of potential for typos and little mistakes (like forgetting a closing tag for a `<p>`). In addition to that, different browsers sometimes treat `innerHTML` in different ways.

If at all possible, you should never change content or presentation directly from your code. Instead, rely on CSS classes to change presentation, and ... what can you use from your code to change structure *without* introducing typos or misplaced end tags? Figure out the answer to that, and you're well on your way to a solution for the exercise above.

We need to add content AND structure to the "currentWord" <div>

When a player clicks on a letter, that letter should be added to the current word. Right now, we've got a `<div>` with an id of "currentWord," but nothing in that `<div>`:

```
<div id="currentWord"></div>
```

So what do we need? Well, we've got to insert text in that `<div>`, but text doesn't usually go directly inside a `<div>`. Text belongs in a textual element, like a `<p>`. So what we really want is something more like this:

```
<div id="currentWord">
  <p>Current Word</p>
</div>
```

Use the DOM to change a page's structure

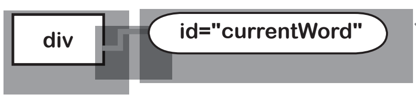
You already know that using code like this isn't that great of an idea:

```
var currentWordDiv = getElementById("currentWord");
currentWordDiv.innerHTML = "<p>" + tileLetter + "</p>";
```

Besides this being a hotbed for typos, what do you do afterward to get the existing current word and append to it?

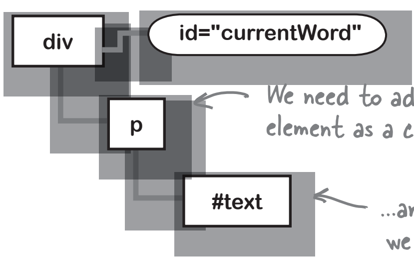
But there's a way to work with the structure of a page without using `innerHTML`: the DOM. You've already used the DOM to get around on a page, but you can use the DOM to ***change*** a page, too.

From the browser's point of view, here's the part of the DOM tree representing the currentWord `<div>`:



The DOM sees the <div> as an element node named div with an id attribute having a value of "currentWord."

We need to create something that looks more like this:

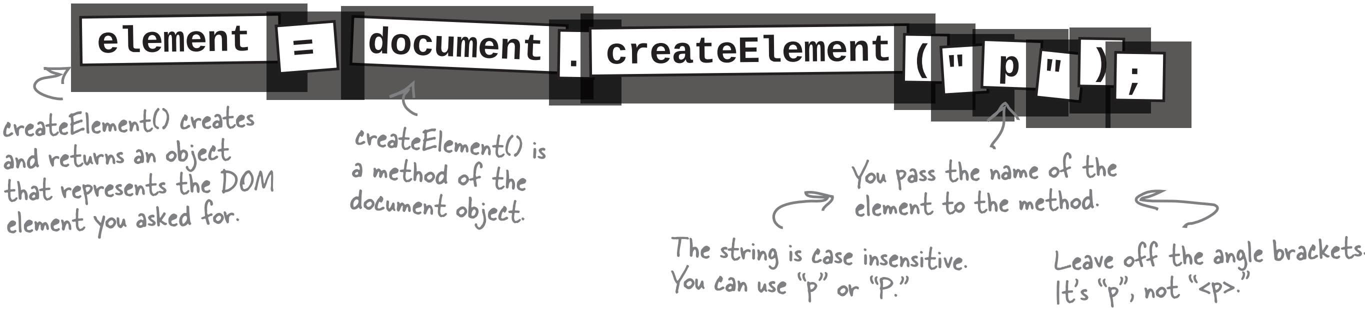


We need to add a new <p> element as a child of the <div>...

...and a new text node where we can add letters as the user clicks on tiles.

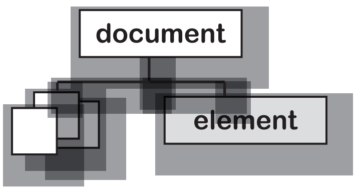
Use createElement() to create a DOM element

Remember the `document` object? We're going to use it again. You can call `document.createElement()` to create a new element. Just give the `createElement()` method the name of the element to create, like "p" or "img":

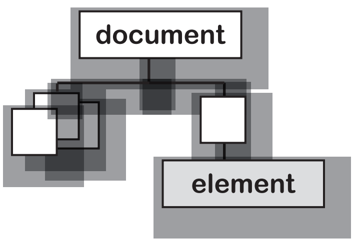


Sharpen your pencil

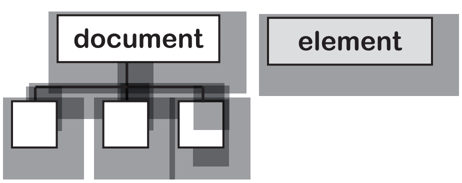
The createElement() method is part of the document element, which contains everything else in the browser's DOM tree. Where do you think the new element is added in the DOM tree?



☐ It's a child of the document element at the top of the DOM tree.



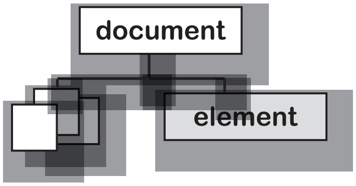
☐ It's a leaf node at the bottom of the DOM tree.



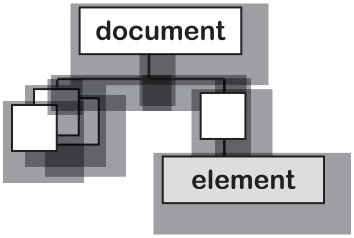
☐ Nowhere. The new element doesn't become part of the tree.

Sharpen your pencil Solution

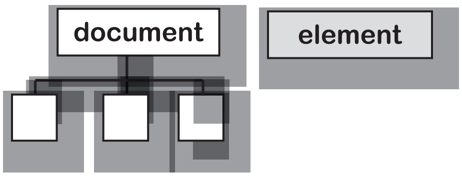
The createElement() method is part of the document element, which contains everything else in the browser's DOM tree. Where do you think the new element is added in the DOM tree?



- [ ] It's a child of the document element at the top of the DOM tree.



- [ ] It's a leaf node at the bottom of the DOM tree.



- [x] Nowhere. The new element doesn't become part of the tree.

You have to TELL the browser where to put any new DOM nodes you create

Web browsers are good at following directions, but they're not so great at figuring things out on their own. When you create a new node, the browser has no idea where that node should go. So it just holds on to it until you ***tell*** the browser where the node goes.

That works out pretty well, too, because you already know how to add a new child node to an element: with `appendChild()`. So we can create a new element, and then append it to an existing element as a new child:

```
var currentWordDiv = getElementById("currentWord");
var p = document.createElement("p");
currentWordDiv.appendChild(p);
```

This gets the <div> we'll want to use as a parent.

Then we create a new <p> element.

Finally, we add the <p> as a child of the <div>.



You can create elements, text, attributes, and a lot more.

The `document` object has all sorts of helpful create methods. You can `createElement()`, `createTextNode()`, `createAttribute()`, and a lot more.

Each method returns a new node, and you can insert that node anywhere into your DOM tree you want. Just remember, until you insert the node into the DOM tree, it won't appear on your page.

```
text = document.createTextNode("Hello!");
```

All the create methods return a new node.

createTextNode() takes the text for the new node as its single argument.

```
att = document.createAttribute("id","tile21");
```

An attribute node belongs on an element... that's up to you to take care of, though.

createAttribute() takes an attribute name and value as arguments.

Exercise

See if you can complete the code for adding a letter to the currentWord <div>. Add your code into the addLetter() event handler, and try things out. Does everything work like you expected?



Your job was to finish up adding letters to the currentWord <div>. Were you able to finish the code up? Were there any problems?

```
function addLetter() {
  var tileClasses = this.className.split(" ");
  var letterClass = tileClasses[2];
  var tileLetter = letterClass.substring(1,2);

  var currentWordDiv = document.getElementById("currentWord");
  var p = document.createElement("p");
  currentWordDiv.appendChild(p);
  var letterText = document.createTextNode(tileLetter);
  p.appendChild(letterText);
}
```

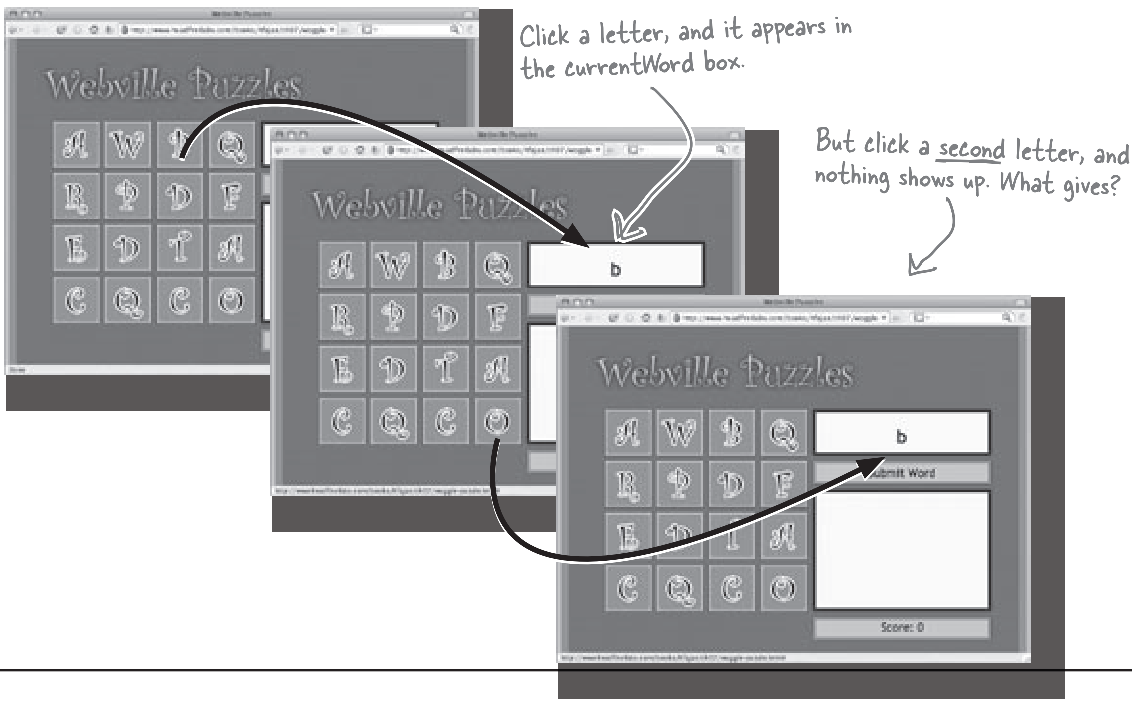
Get the right <div>...

...create and add a <p>...

...and then create and add the letter to that <p>.



But it only works the first time!





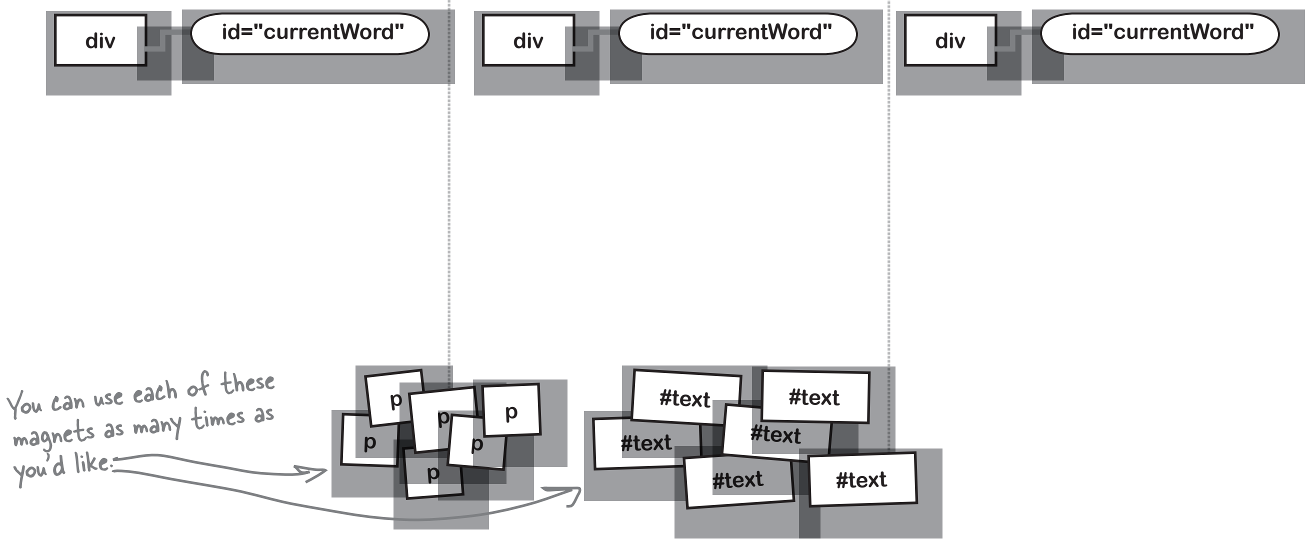
DOM Magnets

Let's try and figure out what's going on with Woggle and our addLetter() event handler. Use the DOM magnets below to build the DOM tree under the currentWord <div> for:

...the first time addLetter() is called:

...the second time addLetter() is called:

...the third time addLetter() is called:



there are no Dumb Questions

Q: When I call appendChild(), where exactly is the node I pass into that method added?

A: `appendChild()` adds a node as the *last* child of the parent element.

Q: What if I don't want the new node to be the last child?

A: You can use the `insertBefore()` method. You pass `insertBefore()` *two* nodes: the node to add, and an existing node that the new node should *precede*.

Q: Didn't we use appendChild() to move elements in the last chapter?

A: We sure did. Whatever node is passed to `appendChild()`, or `insertBefore()`, is added as a new child node to the parent you call `appendChild()` on. The browser moves the node if it's already in the DOM tree, or adds the node into the DOM tree if it's not part of the tree already.

Q: What happens when you append or insert a node that already has children of its own?

A: The browser inserts the element you insert *and all of its children* into the DOM tree. So when you move a node, you're moving that node and everything underneath that node in the DOM tree.

Q: Can you just remove a node from a DOM tree?

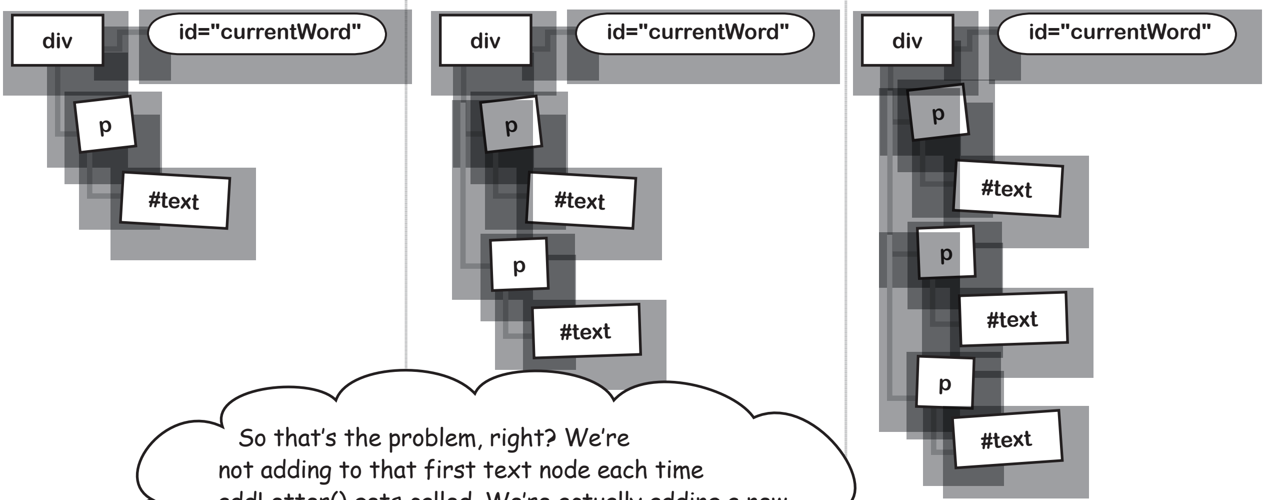
A: Yes, you can use the `removeNode()` methods to remove a node completely from a DOM tree.

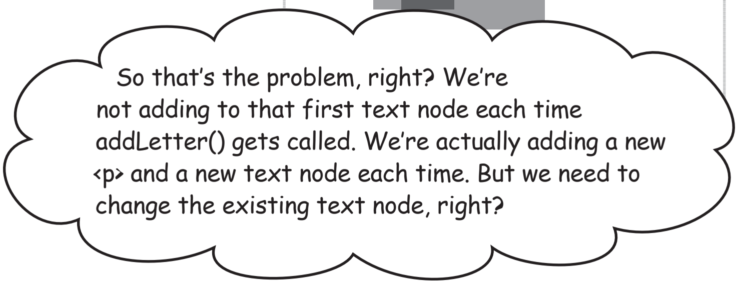


DOM Magnet Solutions

Let's try and figure out what's going on with Woggle and our addLetter() event handler. Use the DOM magnets below to build the DOM tree under the currentWord <div> for:

...the first time addLetter() is called: | **...the second time addLetter() is called:** | **...the third time addLetter() is called:**

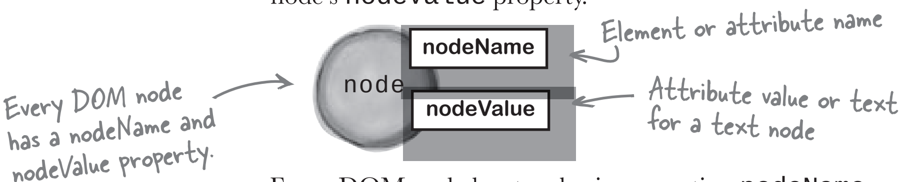






Some nodes have a nodeName, others have a nodeValue, and still others have both.

The first time `addLetter()` gets called, we create a new text node. But on future calls, we need `addLetter()` to change the text in that node. We can do that using the text node's `nodeValue` property.



Every DOM node has two basic properties: `nodeName` and `nodeValue`. For an element, the `nodeName` is the name of the element. For an attribute, `nodeName` is the attribute name, and `nodeValue` is the attribute value. And for a text node, the `nodeValue` is the text in the node.

The nodeName and nodeValue properties tell you about nodes

node	node type	nodeName	nodeValue
document	document node	"document"	null
head	element node	"head"	null
p	element node	"p"	null
Webville Puzzles	text node	"#text"	"Webville Puzzles"
id	attribute node	"id"	"letterbox"

The nodeName of a node is the same as its tag

element nodes never have a nodeValue

The nodeValue of a text node is its text

Sharpen your pencil

You're ready to finish up the section of addLetter() that gets a letter from a clicked-on tile, and adds the clicked-on letter to the currentWord <div>. See if you can write the rest of the function now... and don't forget to test things out!

Write your version of addLetter() here.

Hint: node.childNodes returns an array of a node's children, and node.childNodes.length tells you how many children a node has.

Sharpen your pencil Solution

Using what you learned about the DOM in the last two chapters, were you able to finish up this section of addLetter()? You should have come up with something like this:

```
function addLetter() {
  var tileClasses = this.className.split(" ");
  var letterClass = tileClasses[2];
  var tileLetter = letterClass.substring(1,2);

  var currentWordDiv = document.getElementById("currentWord");

  if (currentWordDiv.childNodes.length == 0) {

    var p = document.createElement("p");

    currentWordDiv.appendChild(p);

    var letterText = document.createTextNode(tileLetter);

    p.appendChild(letterText);

  } else {

    var p = currentWordDiv.firstChild;

    var letterText = p.firstChild;

    letterText.nodeValue += tileLetter;

  }
}
```

The first thing we need to do is see if the currentWord <div> has any children already. (→ `if (currentWordDiv.childNodes.length == 0)`)

This is the code we had before. Now this code only runs when the currentWord <div> has no child nodes.

If the currentWord <div> has children... (→ `} else {`)

...we can get the <p>, and then the text node child of that <p>...

...and add the new letter to the text node.

there are no Dumb Questions

Q: What is that childNodes property again?

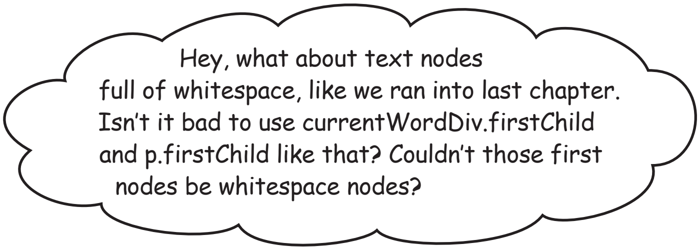
A: `childNodes` is a property available on every node. It returns an array of all of a node's children, or `null` if there aren't any children for that node. And since it's an array, it has a `length` property that tells you how many nodes are in the array.

Q: Can't I just keep up with whether or not addLetter() has been called, and use that as my conditional?

A: No, that won't always work. It's true that the first time `addLetter()` is called, you need to create a `<p>` and text node. But if the player submits a word, and the board is reset, `addLetter()` would again need to create a new `<p>` and text node. So just checking to see how many times `addLetter()` has run won't be enough.

Q: I wrote my code a different way. Is that okay?

A: Sure. There are usually at least two or three different ways to solve a problem. But you need to be sure that your code always works... and that it's not creating DOM nodes unless it needs to. If both of those things are true, then feel free to use your own version of `addLetter()`.

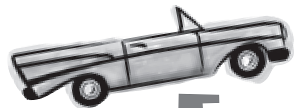




When you control the DOM structure, nothing happens that you don't specify.

When the browser creates a DOM tree based on an XHTML text file, ***the browser is in control***. It's doing what it thinks is best to represent that XHTML - and sometimes, that means interpreting line endings or extra tabs and spaces as text nodes filled with whitespace.

But when you're making changes to a DOM tree, ***you're*** the one in control. The browser won't insert anything unless you tell it to. So when you're working with the DOM nodes you inserted into the currentWord `<div>`, you don't have to worry about extra whitespace text nodes. Instead, you can go right to the first child of the `<div>`, and know that it's a `<p>`.



Test Drive

Test out your new-and-improved event handler.

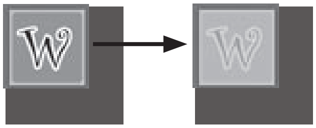
See how `addLetter()` works now. Each time you click a tile, it should add another letter to the current word box. Does it work?



There are a few things we still need to do related to clicking on tiles, though... can you figure out what they are?

We need to disable each tile. That means changing the tile's CSS class...

Once a player's clicked on a tile, they can't re-click on that tile. So we need to disable each tile once it's been clicked.



Clicking on a tile should change the look of that tile.

This is presentation, so you probably already know what to do, don't you? In `addLetter()`, we need to add another CSS class to the clicked-on tile. There's a class in `puzzle.css` called "disabled" that's perfect for the job.

Add this line to the end of `addLetter()`:

```
function addLetter() {
  // existing code

  this.className += " disabled";
}
```

We need to add this class to the existing tile classes, not just replace those classes.

Make sure there's a space to separate the CSS classes from each other.

Now, once `addLetter()` runs, the clicked-on tile fades, and then looks like it can't be clicked anymore.

...AND turning OFF the addLetter() event handler

As long as there have been games, there have been gamers looking for an edge. With Woggle, even though we can disable the ***look*** of a tile, that doesn't mean clicking on the tile doesn't do anything. Clicking on a tile—even if it's got the `disabled` class—will still trigger the `addLetter()` event handler. That means a letter can be clicked on an infinite number of times... unless you put a stop to it!

So we need to take another step at the end of `addLetter()`. We need to remove the `addLetter()` event handler for the clicked-on tile.

```
function addLetter() {
  // existing code

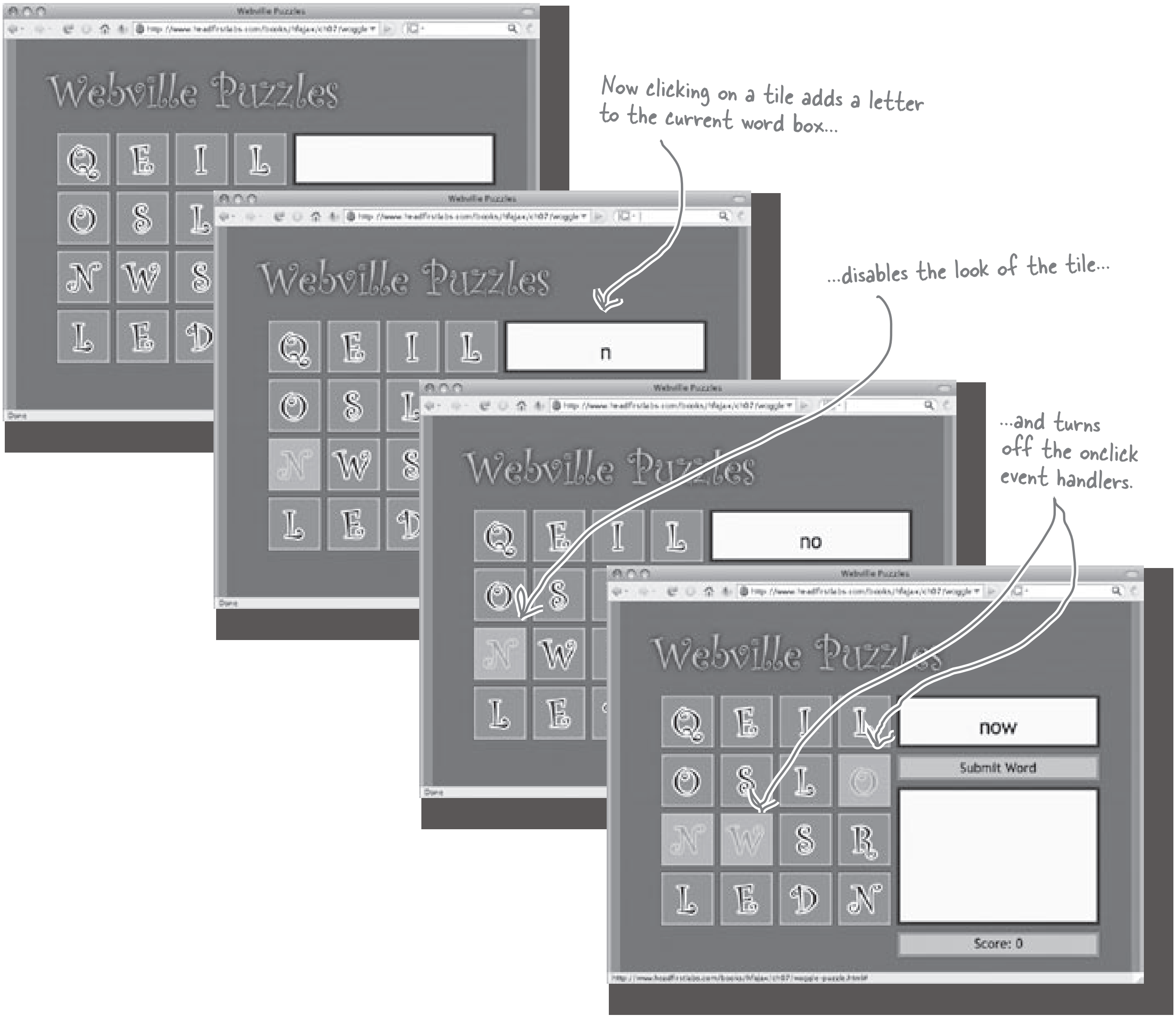
  this.className += " disabled";
  this.onclick = "";
}
```

Set onclick to an empty string. That removes the addLetter() event handler.

Test Drive

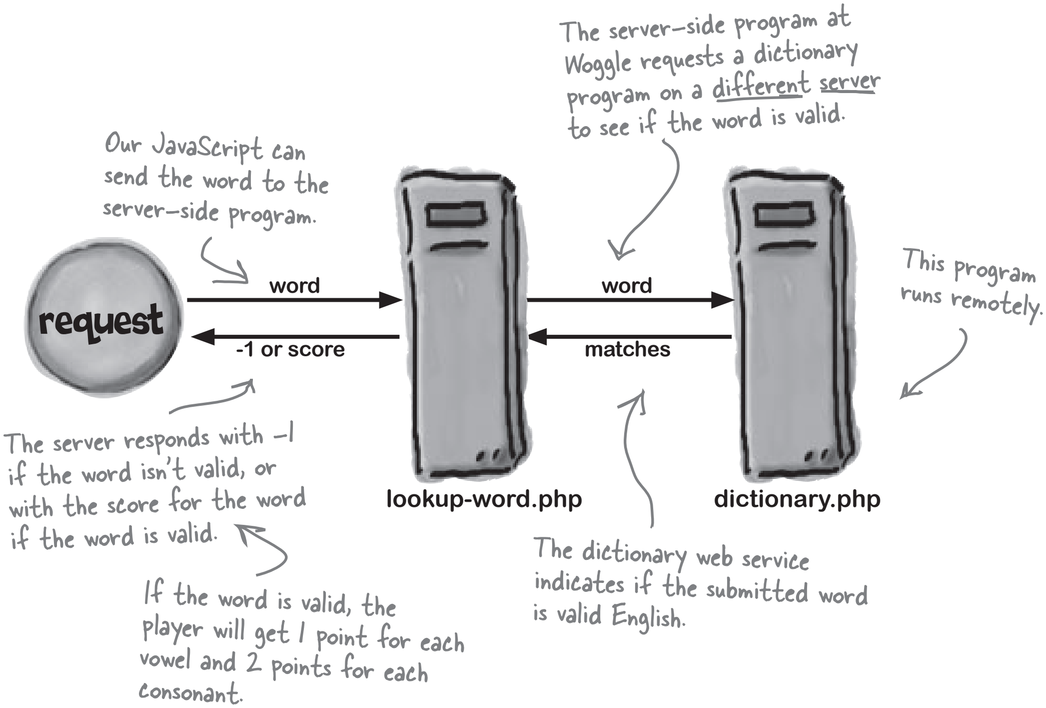
We've handled tile clicks... completely!

Have you made all the additions you need to `addLetter()`? Once you have, fire up Woggle, and build some words.



Submitting a word is just (another) request

`addLetter()` was all about using the DOM, and submitting a word to the server is all about request objects. Woggle's already got a program on their server that takes in a word and returns a score for that word... or a -1 if the word isn't a real English word.



Our JavaScript doesn't care how the server figures out its response to our request

With Woggle, it really doesn't matter that the server-side program we're calling makes *another* request to *another* program. In fact, it wouldn't matter even if `lookup-word.php` called a PHP program, then made a SOAP request to a Java web service, and then sent a message to a cell phone using an SMS gateway. All that *does* matter is that we send the server-side program the right information, and it returns to us the right response.

Your JavaScript only needs to worry about sending requests and handling responses... not how the server gets those responses.

Sharpen your pencil

You've built and sent a lot of request objects by now. Using what you've learned, can you write the submitWord() function?

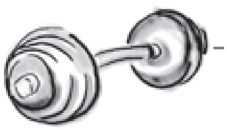
Answers on page 316.

Wait a second... before you start waving around all your fancy asynchronous request objects again, why should we be making an asynchronous request here? Shouldn't players have to **wait** for the server to check their word and return a score?



Not every request should be an ASYNCHRONOUS request.

In all the earlier chapters, we made asynchronous requests, so users weren't stuck waiting on a password to get checked or a page to get loaded. But in Woggle, we really want users to wait on the server before doing anything else. So when a word is sent to the server for scoring, we need to use a ***synchronous*** request.



BRAIN BARBELL

There are some other advantages to using a synchronous request for submitting a word to the server in Woggle. Can you think of any?

So if we're using a synchronous request, do we need a callback function? If the program waits on the server, why not just write the code to handle the server's response right in our submitWord() function?



Synchronous requests don't require a callback function.

When a request is asynchronous, the browser keeps running your code. So after it runs `request.send(null)`, the browser goes to the next line of your sending function. That's usually the end of the function because we want users to be able to keep working with a web page. Then, when the server responds, a callback gets run that can update the page or respond to what the server said.

But with a *synchronous* request, the browser ***waits on the server***. No more code is run until the server comes back with a response. So in that case, we really don't *need* a callback. We can just continue on with the sending function, and ***know that the request object will have the server's response data in it!***

Sharpen your pencil REVISITED

Go back to the code you wrote on page 313, and make a few changes. First, make sure your request is synchronous, and not asynchronous. Then, remove a reference to a callback; we don't need one! Finally, at the end of the function, display the response from the server using an alert box.

Sharpen your pencil Solution

Your job was to build the submitWord() function... and to make sure it works synchronously. What did you come up with?

```
function submitWord() {
  var request = createRequest();
  if (request == null) {
    alert ("Unable to create request object.");
    return;
  }
  var currentWordDiv = document.getElementById("currentWord");
  var userWord = currentWordDiv.firstChild.firstChild.nodeValue;
  var url = "lookup-word.php?word=" + escape(userWord);
  request.onreadystatechange = updateScore;   (struck out)
  request.open("GET", url, false);
  request.send(null);

  alert("Your score is: " + request.responseText);
}
```

This is pretty standard stuff. Make sure you've got utils.js referenced in your XHTML page.

First we get the <div> with the current word...

...and then we want the first child (the <p>), followed by the first child of that (the text node), and then the node value of <u>that</u>.

We send the request like always, but we use "false," making this a <u>synchronous</u> request.

We're sending a synchronous request, so there's no need for a callback function this time.

The code won't get here until the server responds, so it's safe to use the responseText property.

there are no Dumb Questions

Q: I got a little lost on that currentWordDiv.firstChild.firstChild. nodeValue bit. Can you explain that?

A: Sure. You can break that statement down into parts. So first, there's `currentWordDiv.firstChild`. That's the first child of the `<div>`, which is a `<p>`. Then, we get the `firstChild` of that, which is a text node. And finally, we get the `nodeValue` of *that*, which is the text in the node—the word the user entered.

Q: Wow, that's confusing. Do I have to write my code that way?

A: You don't have to, but it's actually a bit faster than breaking things into lots of individual lines. Since this entire statement is parsed at one time, and there's only one variable created, JavaScript will execute this line a bit faster than if you'd broken it into several pieces.

Q: Didn't you forget to check the readyState and status codes of the request object?

A: When you're making a synchronous request, there's no need to check the `readyState` of the request object. The browser won't start running your code again until the server's finished with its response, so the `readyState` would *always* be 4 by the time your code could check it.

You could check the `status` to make sure your request got handled without an error. But since you'll be able to tell that from the actual response, it's often easier to just go right to the `responseText`. Remember, we're not making an asynchronous request. With a synchronous request, there's no need to check `readyState` and `status` in your callback.

Usability check: WHEN can submitWord() get called?

Did you try and test out your new `submitWord()` function? If you did, you probably realized that the function isn't connected to anything. Right now, the "Submit Word" button doesn't do anything. In fact, "Submit Word" is an `<a>` element, and not a button at all!

```
<div id="submit"><a href="#">Submit Word</a></div>
```

This `<div>` and `<a>` are then styled to look like a button on the page. We had a similar situation with the tiles, though, so this shouldn't be a problem. We can assign an event handler to the onclick event of the `<a>` representing the "Submit Word" button:

```
var submitDiv = document.getElementById("submit");
var a = submitDiv.firstChild;
while (a.nodeName == "#text") { a = a.nextSibling; }
a.onclick = submitWord;
```

Get the right <div>.

Get the first child of the <div>.

Since the browser created this part of the DOM, we should make sure we don't have a whitespace text node.

Assign the event handler.

You can't submit a word if there's no word to submit

So where do you think this code goes? In `initPage()`? But that doesn't make sense... in `initPage()`, there aren't any letters in the current word box, so players shouldn't be able to submit anything.

The first time there's a word to submit is the first time there's a letter in the current word box. That's also the first time that a tile is clicked, which turns out to be the first time `addLetter()` is called for a new word.

Fortunately, we've already got a special case: the first time `addLetter()` is called for a new word, we're creating the `<p>` and text node underneath the currentWord `<div>`. So we just need to add the code above to that part of the `addLetter()` event handler:

All of this new code...

...goes right here.

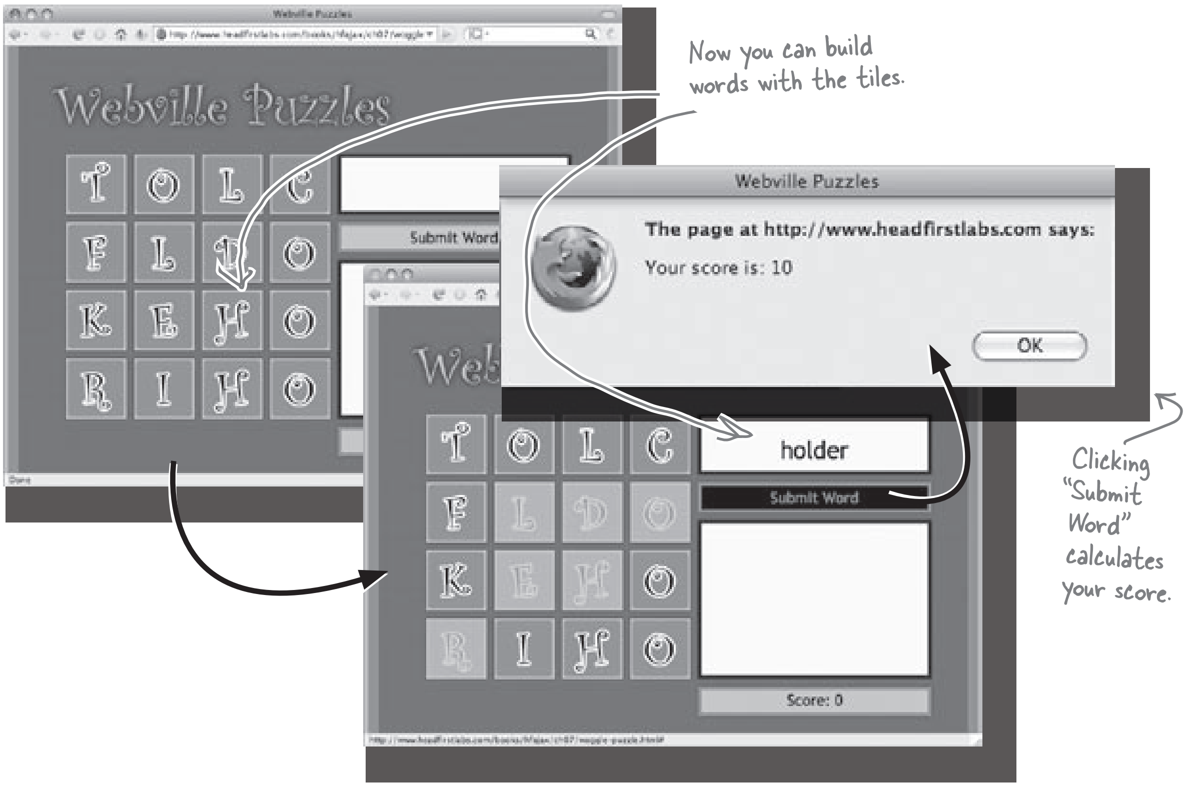
```
if (currentWordDiv.childNodes.length == 0) {
  // existing code to add a new <p> and text node
  // existing code to add in first letter of new word
  // code to enable Submit Word button

} else {  // ... etc ...
```

Test Drive

Get your word score. Go on, get it!

Have you made all the additions you need to `addLetter()`? Once you have, fire up Woggle, and build some words.



Brain Power

Even though clicking "Submit Word" doesn't call submitWord() until at least one letter's entered, "Submit Word" still looks like a button. Can you write code to let users know what they should do if they click "Submit Word" too early?

Who Does What?

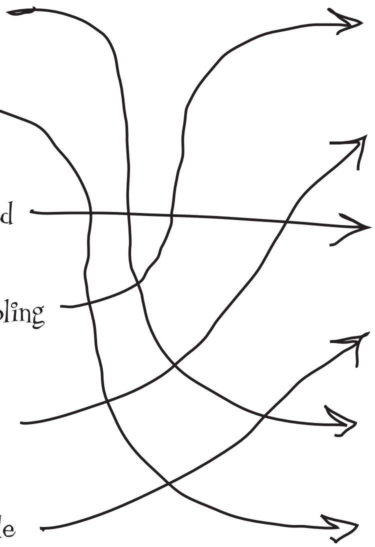
Match the DOM properties and methods on the left to the tasks you'd use those properties and methods to accomplish on the right.

nodeValue	You want the table cell just to the left of the table cell you're in.
parseInt	You want all the <p>'s within a particular <div>.
removeChild	You want to get rid of all the elements on a page.
previousSibling	You want to exchange an <img> element with some descriptive text.
childNodes	You want to print out a name, which is in the <div> with an id of "name."
replaceNode	You need to add the numeric values of two form fields.

Well, this isn't a DOM method, but you never know when you'll need it. (note pointing to parseInt)

WHO DOES WHAT?

Match the DOM properties and methods on the left to the tasks you'd use those properties and methods to accomplish on the right.

nodeValue		You want the table cell just to the left of the table cell you're in.
parseInt		You want all the <p>'s within a particular <div>.
removeChild		You want to get rid of all the elements on a page.
previousSibling		You want to exchange an <img> element with some descriptive text.
childNodes		You want to print out a name, which is in the <div> with an id of "name."
replaceNode		You need to add the numeric values of two form fields.

childNodes gives you the children of a particular DOM node.

When you're replacing one node with a different node, use replaceNode().

Text in a node is represented by the nodeValue property on a text node.

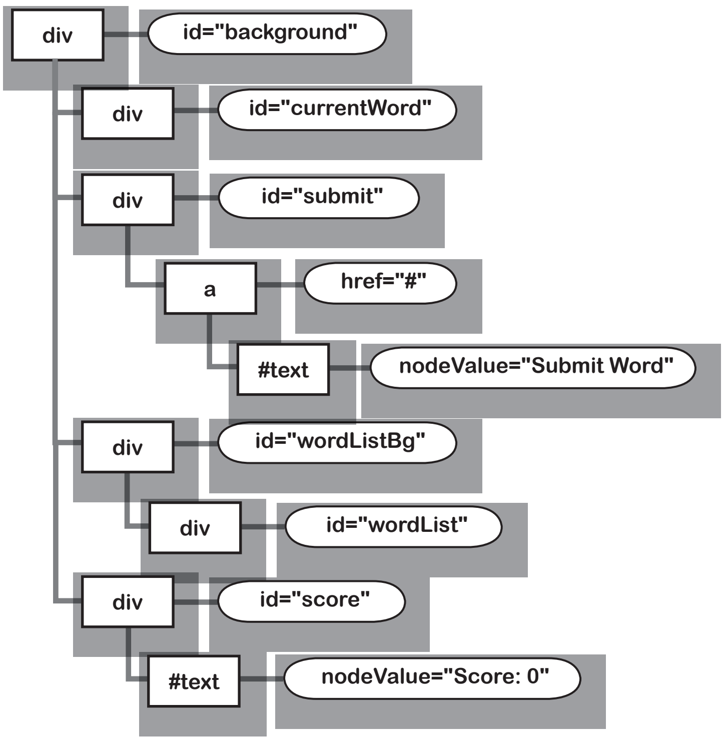
parseInt("21") converts the string "21" into the int 21.



Long Exercise

It's time to put everything you've learned so far to use: DOM manipulation, creating DOM nodes, JavaScript string functions, handling the request from a server... this exercise has it all. Follow the directions below, and tick off the boxes as you complete each step.

- ☐ If the server rejects the submitted word, let the player know with a message that reads, "You have entered an invalid word. Try again!"
- ☐ If the server accepts the submitted word, add the accepted word to the box of accepted words just below the "Submit Word" button.
- ☐ Get the current score, and add the score that the server returns for the just-accepted word. Using this new score, update the "Score: 0" text on the screen.
- ☐ Whether the server accepts or rejects the word, remove the current word from the current word box.
- ☐ Enable all the tiles on the playing board, and reset the "Submit Word" button to its original state.
- ☐ Below is the DOM tree for the sections of the page you're working with, as the browser initially creates the tree. Draw what the DOM tree will look like after your code has run for two accepted words (the specific two words don't matter, as long as the server accepted both of them).





Long Exercise Solution

Below is the completed version of submitWord(). Now it not only submits a word, but updates the score on the page. How close if your solution to ours?

```
function submitWord() {
  var request = createRequest();
  if (request == null) {
    alert ("Unable to create request object.");
    return;
  }
  var currentWordDiv = document.getElementById("currentWord");
  var userWord = currentWordDiv.firstChild.firstChild.nodeValue;
  var url = "lookup-word.php?word=" + escape(userWord);
  request.open("GET", url, false);
  request.send(null);
```

The server returns –1 if the submitted word is invalid.

☑ If the server rejects the submitted word, let the player know.

```
  if (request.responseText == -1) {
    alert("You have entered an invalid word. Try again!");
  } else {
    var wordListDiv = document.getElementById("wordList");
    var p = document.createElement("p");
    var newWord = document.createTextNode(userWord);
    p.appendChild(newWord);
    wordListDiv.appendChild(p);
```

☑ Add the accepted word to the box of accepted words.

This creates a new <p>, a new text node with the user's word, and then adds both to the wordList <div>.

```
    var scoreDiv = document.getElementById("score");
    var scoreNode = scoreDiv.firstChild;
    var scoreText = scoreNode.nodeValue;
    var pieces = scoreText.split(" ");
    var currentScore = parseInt(pieces[1]);
    currentScore += parseInt(request.responseText);
    scoreNode.nodeValue = "Score: " + currentScore;
  }
```

☑ Update the "Score: 0" text on the screen.

You can split "Score: 0" into two parts using split(" ").

We want the second part, and we want it as an int.

Add the server's response, and then update the text node.

```
  var currentWordP = currentWordDiv.firstChild;
  currentWordDiv.removeChild(currentWordP);
  enableAllTiles();
  var submitDiv = document.getElementById("submit");
  var a = submitDiv.firstChild;
  while (a.nodeName == "#text") {
    a = a.nextSibling;
  }
  a.onclick = function() {
    alert("Please click tiles to add letters and create a word.");
  };
}

function enableAllTiles() {
  tiles = document.getElementById("letterbox").getElementsByTagName("a");
  for (i=0; i<tiles.length; i++) {
    var tileClasses = tiles[i].className.split(" ");
    if (tileClasses.length == 4) {
      var newClass =
        tileClasses[0] + " " + tileClasses[1] + " " + tileClasses[2];
      tiles[i].className = newClass;
      tiles[i].onclick = addLetter;
    }
  }
}
```

☑ Remove the current word from the word box.

☑ Enable all the tiles.

Remember to reset the "Submit Word" button to an alert() function for the event handler.

We built a utility function for enabling all the tiles.

A tile that has 4 classes has the "disabled" class at the end.

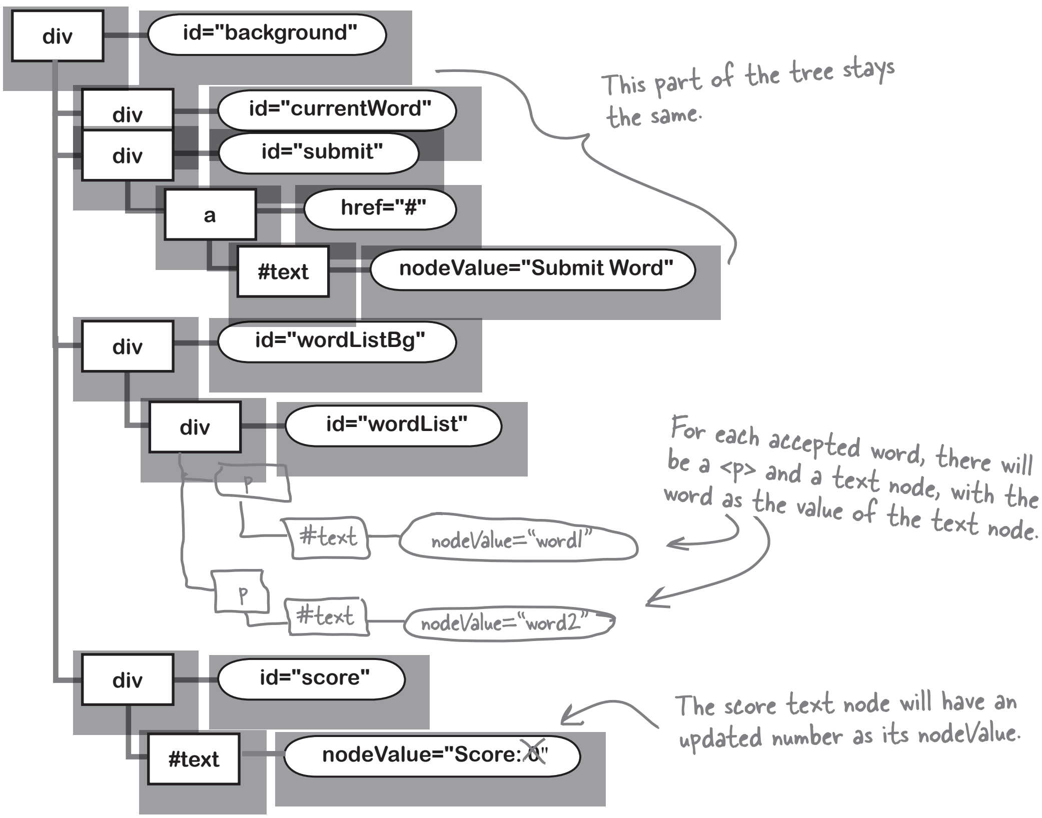
We use the first three existing classes, but drop the fourth.

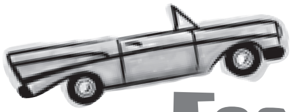
Remember to reset the event handler to addLetter.

Solution continues on the next page.

Long Exercise Solution (continued)

Below is the DOM tree for the sections of the page you're working with, as the browser initially creates the tree. Draw what the DOM tree will look like after your code has run for two accepted words (the specific two words don't matter, as long as the server accepted both of them).

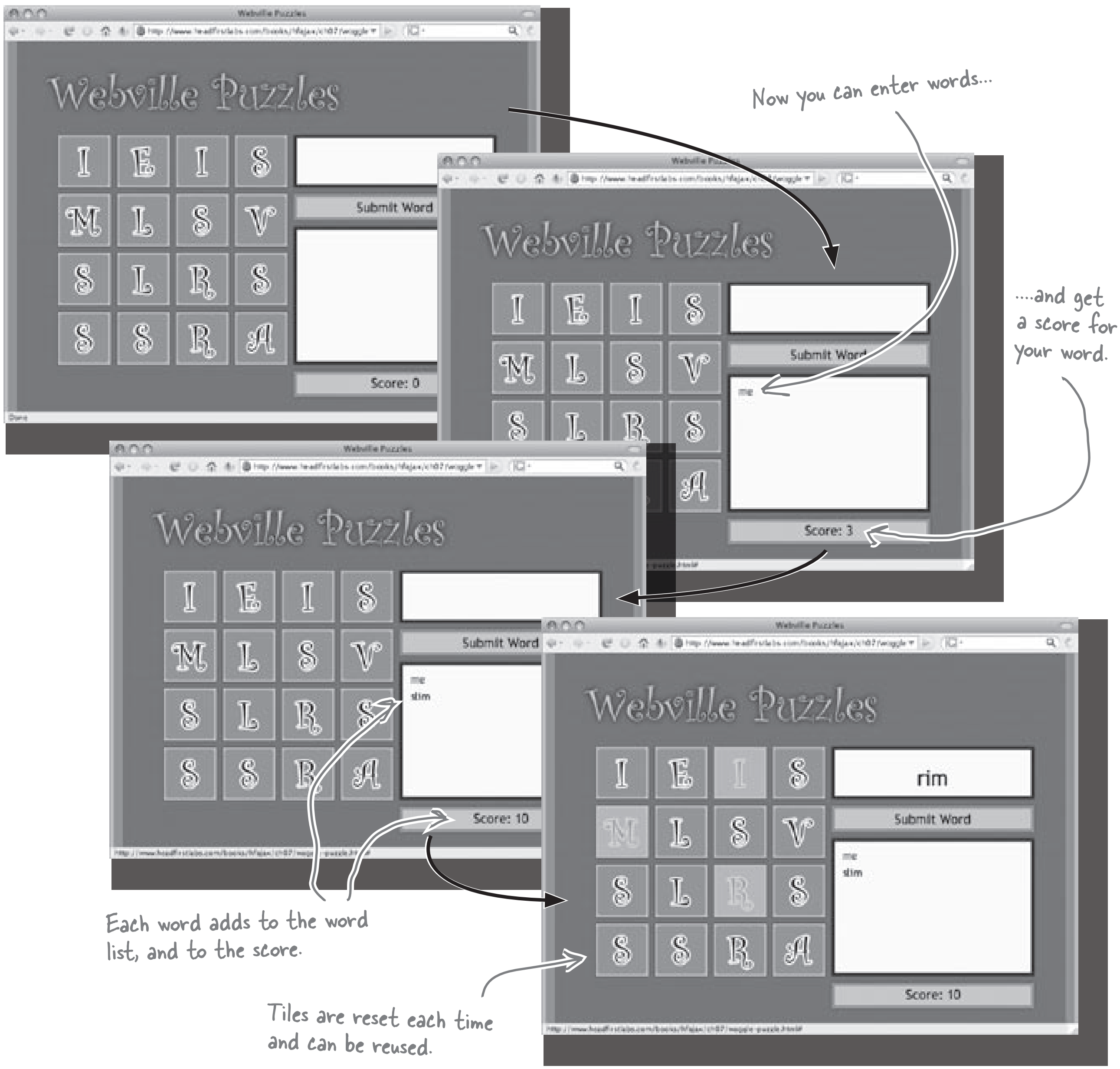




Test Drive

Anyone for Woggle?

Do you have everything working? Try out Woggle... it's all working just the way we imagined way back on page 286.

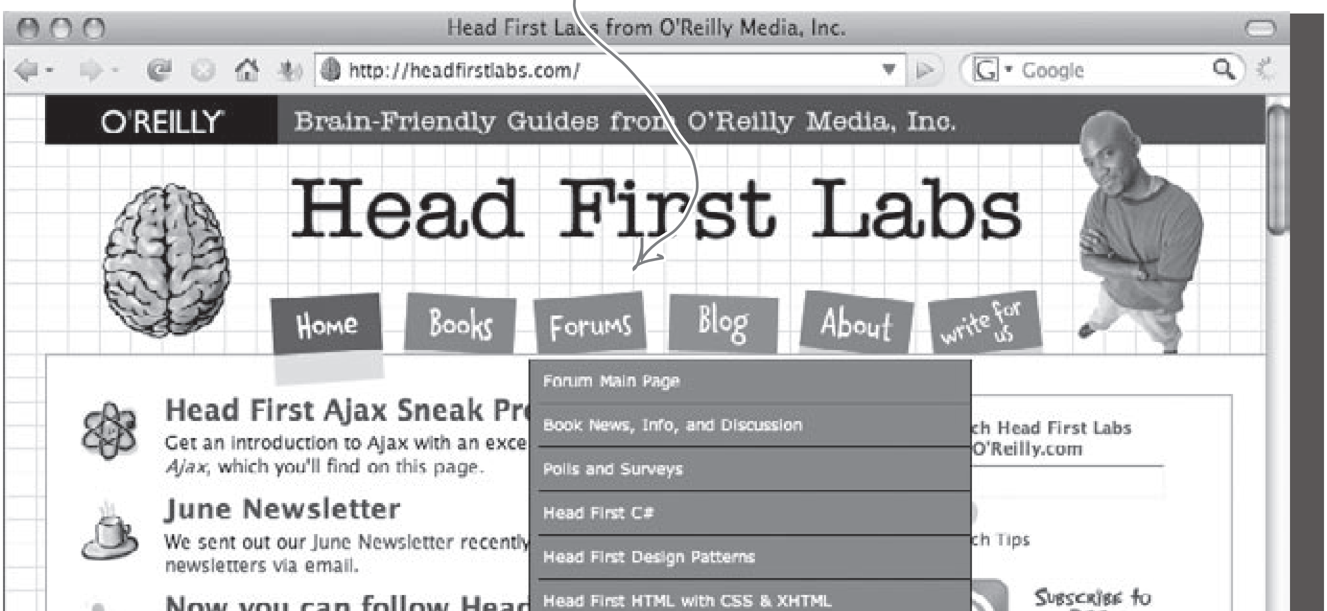


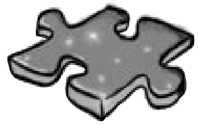
But there's still more to do!

- What if there was a timer that gave you 60 seconds to enter as many words as you could think of?
- What if you could choose a lettered tile, and then only choose tiles next to the last selected tile?
- What if once you used letters to make a valid word, those tiles were replaced by new random tiles?
- And besides all that, how do YOU think Woggle could be improved?

We want to see you put your DOM, JavaScript, and Ajax skills to work. Build your BEST version of Woggle, and submit your URL in the Head First Labs "Head First Ajax" forum. We'll be giving away cool prizes for the best entries in the coming months.

Click here to go to the forums and tell us how to access your version of Woggle.





DOMAcrostic

Take some time to sit back and give your right brain something to do. Answer the questions in the top, then use the letters to fill in the secret message.

This method creates an element of the specified type:

___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
1 2 3 4 5 6 7 8 9 10 12 13 14

This is the name of the game we built in this chapter:

___ ___ ___ ___ ___ ___
15 16 17 18 19 20

This method adds an element to the DOM tree:

___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
21 22 23 24 25 26 27 28 29 30 31

A DOM tree is a just collection of these:

___ ___ ___ ___ ___ ___ ___
32 33 34 35 36 37 38

This method substitutes one node for another:

___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
39 40 41 42 43 44 45 46 47 48 49 50

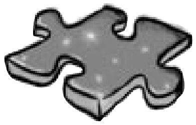
This method removes a node from the DOM tree:

___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
51 52 53 54 55 56 57 58 59 60 61

___ ___ ___   ___ ___ ___ ___ ___ ___ ___
37 58 3   33 39 16 15 38 45 51

___ ___ ___ ___ ___ ___ ___ ___ ___ ___   ___ ___ ___
5 2 21 25 38 8 43 14 45 38   26 32 10

___ ___ ___ ___ ___   ___ ___ ___ ___   ___ ___ ___ ___ ___ ___ ___
13 16 50 52 38   59 13 37 16   54 33 34 9 57 5 38



DOMAcrostic

Take some time to sit back and give your right brain something to do. Answer the questions in the top, then use the letters to fill in the secret message.

This method creates an element of the specified type:

C	R	E	A	T	E	E	L	E	M	E	N	T
1	2	3	4	5	6	7	8	9	10	12	13	14

This is the name of the game we built in this chapter:

W	O	G	G	L	E
15	16	17	18	19	20

This method adds an element to the DOM tree:

A	P	P	E	N	D	C	H	I	L	D
21	22	23	24	25	26	27	28	29	30	31

A DOM tree is a just collection of these:

O	B	J	E	C	T	S
32	33	34	35	36	37	38

This method substitutes one node for another:

R	E	P	L	A	C	E	C	H	I	L	D
39	40	41	42	43	44	45	46	47	48	49	50

This method removes a node from the DOM tree:

R	E	M	O	V	E	C	H	I	L	D
51	52	53	54	55	56	57	58	59	60	61

T	H	E		B	R	O	W	S	E	R
37	58	3		33	39	16	15	38	45	51

T	R	A	N	S	L	A	T	E	S		D	O	M
5	2	21	25	38	8	43	14	45	38		26	32	10

N	O	D	E	S		I	N	T	O		O	B	J	E	C	T	S
13	16	50	52	38		59	13	37	16		54	33	34	9	57	5	38

8 frameworks and toolkits

Trust No One



So what's the *real* story behind all those Ajax frameworks?

If you've been in Webville awhile, you've probably run across at least one JavaScript or Ajax framework. Some frameworks give you **convenience methods for working with the DOM**. Others make **validation** and **sending requests** simple. Still others come with libraries of pre-packaged JavaScript **screen effects**. But which one should you use? And how do you know what's really going on inside that framework? It's time to do more than use other people's code... it's time to ***take control of your applications***.



There are a LOT of options for frameworks that let you work with Ajax in different (and sometimes easier) ways.

If you Google the Internet for "JavaScript framework" or "Ajax library," you'll get a whole slew of links to different toolkits. And each framework's a bit different. Some are great for providing slick screen effects, like drag-and-drop, fades, and transitions. Others are good at sending and receiving Ajax requests in just a line or two of code.

In fact, you've been using a framework of sorts every time you reference a function from `utils.js`. All that script does is provide common functionality in a reusable package. Of course, most frameworks have a lot more functionality, but the principle is still the same.

So which framework should you use? Even more importantly... should you use one at all?

Sharpen your pencil

Deciding to use a JavaScript framework for writing your code is a big deal. Below, write down three reasons that you think it would be a good idea to use a framework... and three reasons you think it might ***not*** be a good idea.

Reasons to use a framework

1. ....

2. ....

3. ....

Reasons NOT to use a framework

1. ....

2. ....

3. ....

there are no Dumb Questions

Q: I don't even know what a framework is. How am I supposed to answer these questions?

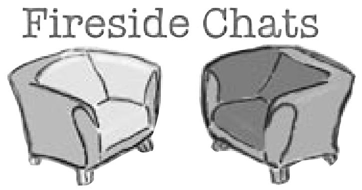
A: A framework is just a JavaScript file—or set of files—that has functions, objects, and methods that you can use in your code. Think of a framework like a bigger, more complete version of the `utils.js` file we've been using.

Q: But I've never used one before!

A: That's okay. Just think about reasons you might like to try out a framework, and what advantages that framework might have over doing things the way you've been doing them so far. Then, think about what you like about how you've been writing code so far... those are reasons you might not use a framework.

Q: Is there a difference between a framework and a toolkit?

A: Not really. Framework and toolkit are used pretty interchangeably in the JavaScript world. Some people will tell you a framework is a structure for how you write all your code, while a toolkit is a collection of utility functions. But that's a distinction that not every framework or toolkit makes, so it's not worth getting hung up on.



Tonight's talk: **Ajax Framework and Do-It-Myself JavaScript go head-to-head on utility functions, toolkits, and the pros and cons of do-it-yourself thinking.**

Ajax Framework: Wow, I thought you guys were never going to have me on. What is this, like page 332 or something, and I'm just now making an appearance?

Do-It-Myself JavaScript: Hey, we figured you'd show up when you were needed. And lookie here, seven chapters down, and you're just now getting involved.

Ajax Framework: Oh boy. Here we go... you're one of these JavaScript purists, aren't you? No frameworks, no utility functions, just hard work and thousands of lines of code in a single `.js` file, am I right?

Do-It-Myself JavaScript: Not at all. In fact, I'm a big fan of abstracting common code into utility methods, not writing duplicate code, and even having different `.js` files for different sets of functionality.

Ajax Framework: So what's your problem with me? I'd think a guy like you would love me. I take all those routine, boring, annoying tasks and wrap them up into user-friendly function and method calls.

Do-It-Myself JavaScript: Well, that's just it. You wrap them up... you don't abstract them into a different file. You actually hide those details away.

Ajax Framework: And? What's the problem with that? I don't even see the difference... wrapping? abstracting?

Do-It-Myself JavaScript: Hey, you can always look at my code. Just open another script, and you know exactly what's going on. No mystery, no "magic function." That's me, alright.

Ajax Framework: You're kidding, right? I'm just JavaScript, too. You can open me up anytime you want. So how is my JavaScript `.js` file any different than yours?

Ajax Framework: | **Do-It-Myself JavaScript:**

Do-It-Myself JavaScript: Have you ever looked at yourself? Maybe in a mirror, or in the reflection from one of those bright shiny widgets you're so proud of?

Ajax Framework: Oh, all the time. What's your point, Mr. Heavy-Handed?

Do-It-Myself JavaScript: You're impossible to figure out! There's like a thousand lines of code to wade through. What if I want to do something just a bit *differently* than you're set up for? What then?

Ajax Framework: I've got options, man. Tons of options. Sometimes almost a hundred for certain methods. Beat that!

Do-It-Myself JavaScript: Why in the world would I want to? Who wants to figure out what the eighth parameter to a method is? Since when is that helpful?

Ajax Framework: Uhhh... gee, lemme think... well, how about when you don't know how to do what you need to do yourself? Ever tried to code drag-and-drop? Or move around within an image, zooming in and zooming out? You want to build all that yourself?

Do-It-Myself JavaScript: If that's what it takes to actually understand what's going on, you bet I do!

Ajax Framework: Hey, we're not talking about atomic fusion here. Sometimes you just need to get some little visual effect done... or an Ajax request sent. That's no time to be digging around on the Internet for some code a junior high dropout posted to his blog three years ago.

Do-It-Myself JavaScript: I'll bet that kid knows what he's ***doing***, though!

Ajax Framework: Yeah, and he's also driving a '76 Pinto 'cause no one will ***hire*** him. Because he's so ***slow*** at writing basic code!

Do-It-Myself JavaScript: *Whatever.*

Sharpen your pencil Solution

Your job was to think of some good reasons to use a framework, and some not-so-good reasons that come with using a framework. What did you write down?

Reasons to use a framework

1. You don't have to write code for functions that someone else has already figured out. You can just use the existing code in frameworks.
2. Frameworks have functions you might not have time to write yourself but would use if those functions were available. So you get more functionality.
3. The code in frameworks is tested more because more people are using the framework. So there's less chance of bugs, and less need for testing.
4. Frameworks usually take care of cross-browser issues for you, so you don't have to worry about IE, or Firefox, or Opera.

We only asked for three, but we couldn't resist adding this one. It's a BIG reason for using frameworks.

Reasons NOT to use a framework

1. You don't really know what the framework's doing. It might be doing things well... or it might be doing them more inefficiently than you would.
2. The framework might not have all the options you want or need. So you might end up changing your code to accommodate the framework.
3. Sometimes a framework hides important concepts that would be helpful to know. So you might not learn as much using a framework.

BRAIN POWER

Are there certain categories of functionality that you think would be better suited for a framework? What about things you ***don't*** want a framework doing for you?

So what frameworks ARE there?

There are several popular frameworks out there... and most of them do a few different things. Here's the ones that most people are buzzing about:

Prototype (http://www.prototypejs.org)

Prototype is a workhorse library. It provides lots of low-level JavaScript utilities, including support for Ajax.

jQuery is one of the most popular toolkits for JavaScript programming, including Ajax requests.

jQuery (http://www.jquery.com)

script.aculo.us is an add-on to Prototype, and is aimed at providing screen effects in JavaScript.

mooTools (http://mootools.net)

script.aculo.us (http://script.aculo.us)

mooTools is newer, but very full-featured. You get screen effects and Ajax request utilities.

How often do these things change? Are frameworks releasing new versions very often? Is that something we should be worried about?



Frameworks usually change FASTER than the underlying JavaScript syntax does.

Frameworks are controlled by the people who write them, and so a framework might release a new version every few months... or in early stages, every few weeks! In fact, a framework might lose popularity and totally disappear over the course of six or seven months.

But the core JavaScript syntax and objects, like XMLHttpRequest and the DOM, are controlled by big, slow-moving standards groups. So that sort of syntax won't change very often. At the most, you'll see something change every few ***years***.

Every framework uses a different syntax to do things

Each framework uses a different syntax to get things done. For example, here's how you'd make a request and specify what to do with the server's response in Prototype:

```
function checkUsername() {
  var usernameObj = $("username");
  usernameObj.className = "thinking";
  var username = escape(usernameObj.value);
  new Ajax.Request("checkName.php", {
    method:"get",
    parameters: "username=" + username,
    onSuccess: function(transport){
      if (transport.responseText == "okay") {
        $("username").className = "approved";
        $("register").disabled = false;
      } else {
        var usernameObj = ${"username");
        usernameObj.className = "denied";
        usernameObj.focus();
        usernameObj.select();
        $("register").disabled = true;
      }
    },
    onFailure: function() { alert("Error in validation."); }
  });
}
```

The first part of both bits of code gets a value from the page.

This gets the element with an id of "username."

Then a request is made. This is a lot shorter in Prototype.

This is the Ajax object for making requests in Prototype.

transport is the Prototype "stand-in" for the request object.

You usually give Prototype the callback inline... but it's the same basic code, just a little different syntax.

The onSuccess function runs when the server responds normally.

The onFailure function runs if there's a problem with the request or response.

We haven't been providing an error message if the status code isn't 200, or if other problems occur. Prototype handles this nicely, though.

The syntax may change... but the JavaScript is still the same

At a glance, that Prototype code looks pretty different from anything you've written before. But take a look at the equivalent JavaScript from an early version of Mike's Movies registration page:

This code looks a lot different at first... but it turns out to be very similar to the code you write to use a toolkit.

```
function checkUsername() {
  document.getElementById("username").className = "thinking";
  request = createRequest();
  if (request == null)
    alert("Unable to create request");
  else {
    var theName = document.getElementById("username").value;
    var username = escape(theName);
    var url= "checkName.php?username=" + username;
    request.onreadystatechange = showUsernameStatus;
    request.open("GET", url, true);
    request.send(null);
  }
}
function showUsernameStatus() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      if (request.responseText == "okay") {
        document.getElementById("username").className = "approved";
        document.getElementById("register").disabled = false;
      } else {
        document.getElementById("username").className = "denied";
        document.getElementById("username").focus();
        document.getElementById("username").select();
        document.getElementById("register").disabled = true;
      }
    }
  }
}
```

But that's just the same code we've been writing... it looks a little different, but it's the same stuff! I don't want to learn **another** new set of syntax.



JavaScript and Ajax frameworks are just new and different ways to do what you've already been doing.

When you boil it all down to code, an asynchronous request is an asynchronous request is an asynchronous request. In other words, under the hood, your JavaScript code still has to create a request object, set up code to run based on what the server returns, and then send that request. No matter how the syntax changes, the basic process stays the same.

Using a framework might make parts of setting up and sending that request easier, but a framework won't fundamentally change what you've been doing. And yes, you'll definitely need to learn some new syntax to use any framework effectively.

But I'll bet there are some pretty big advantages, too, right? Like cool visual effects, and maybe some more robust error handling?

Frameworks offer a lot of nice features "for free."

Most frameworks come with a lot of convenience methods and cool visual effects. And the syntax isn't really that hard to pick up if you're already familiar with basic JavaScript and Ajax concepts and principles.

And one of the best features of frameworks is that a lot of them handle situations where users don't have JavaScript enabled in their browsers.



there are no Dumb Questions

Q: You didn't mention my favorite framework, [insert your framework name here]. What's up with that?

A: Well, there are a lot of frameworks out there, and more are showing up every day. The frameworks on page 335 are some of the most popular right now, but your framework might show up on that list in a few months.

In any case, the main thing is that a framework doesn't provide fundamentally different functionality than the code you've been writing. It just makes that functionality more convenient, or it takes less time to write, or it adds visuals... you get the idea.

Q: Do all frameworks make working with elements on a page so easy?

A: If you use a framework, you probably won't be writing a lot of DOM methods, like `getElementById()` or `getElementsByTagName()`. Since those are such common operations, most frameworks provide syntax to make that easier, like `$("username")` to get an element with an id of "username."

Q: So what's the framework using to get the element, then?

A: The same DOM methods you'd use without the framework. `$("username")` just gets turned into a call to `document.getElementById("username")`. Additionally, the returned object has its normal DOM methods available, as well as additional methods the framework might provide.

Q: So frameworks are a good thing, right?

A: Well, some people use frameworks because they don't want to take the time to learn the underlying concepts. That's *not* a good thing because those folks don't really know what's going on in their code.

If *you* use a framework, though, you *do* know the concepts and code underneath. That means you'll probably be more effective as a programmer, and be able to hunt down problems a little more effectively, too.

Q: So a framework just does what we've already been doing ourselves?

A: Well, frameworks often do a little *more* than we've been doing. They typically provide more options, and they also tend to have a more robust error handling setup than just showing an `alert()` box. But they're still making requests and handling responses and grabbing elements on a page using the DOM, just like the code you've been writing.

Q: So we shouldn't use frameworks since we already know how to write all that request and response and DOM code, right?

A: Well, frameworks do offer a lot of convenience functions, and those screen effects are pretty cool...

Q: So then we *should* use frameworks?

A: We didn't say that either. There's a certain amount of control you lose with a framework because it might not do just what you want it to in a certain situation. Sometimes it's best to take complete control, and just write the code you need without putting a framework in the mix.

Q: So which is it? Use a framework, or don't use one?

A: That's the question, isn't it? Turn the page, and let's try and figure that out.

Frameworks can't solve your programming problems for you.

It's up to YOU to UNDERSTAND your code, whether or not you use a framework to make that code easier to write.

To framework or not to framework?

There are a lot of good reasons to use a framework... and plenty of reasons to not use one. Some people go back and forth between projects where they use a framework and projects where they don't. It really depends on the situation and your personal preferences.







I use a toolkit to handle errors, deal with weird mobile browsers, and let my team focus on functionality, not the browser wars.



Business developer building a micro site for mobile devices

We have lots of weird requests that interact with our server programs and the data... I avoid toolkits because I want complete control of how my requests are configured and sent.



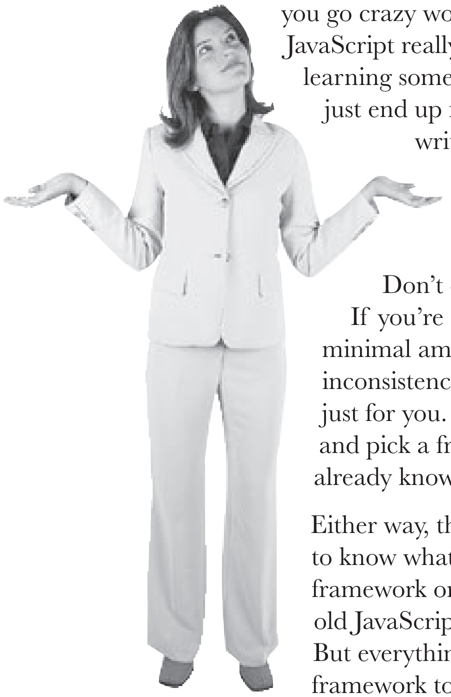
Database engineer at a media company

I think mooTools looks pretty cool, but I want to get a better handle on requests and callbacks. I'll probably use a toolkit at some point, but just not yet.



Junior software developer

The choice is up to you...



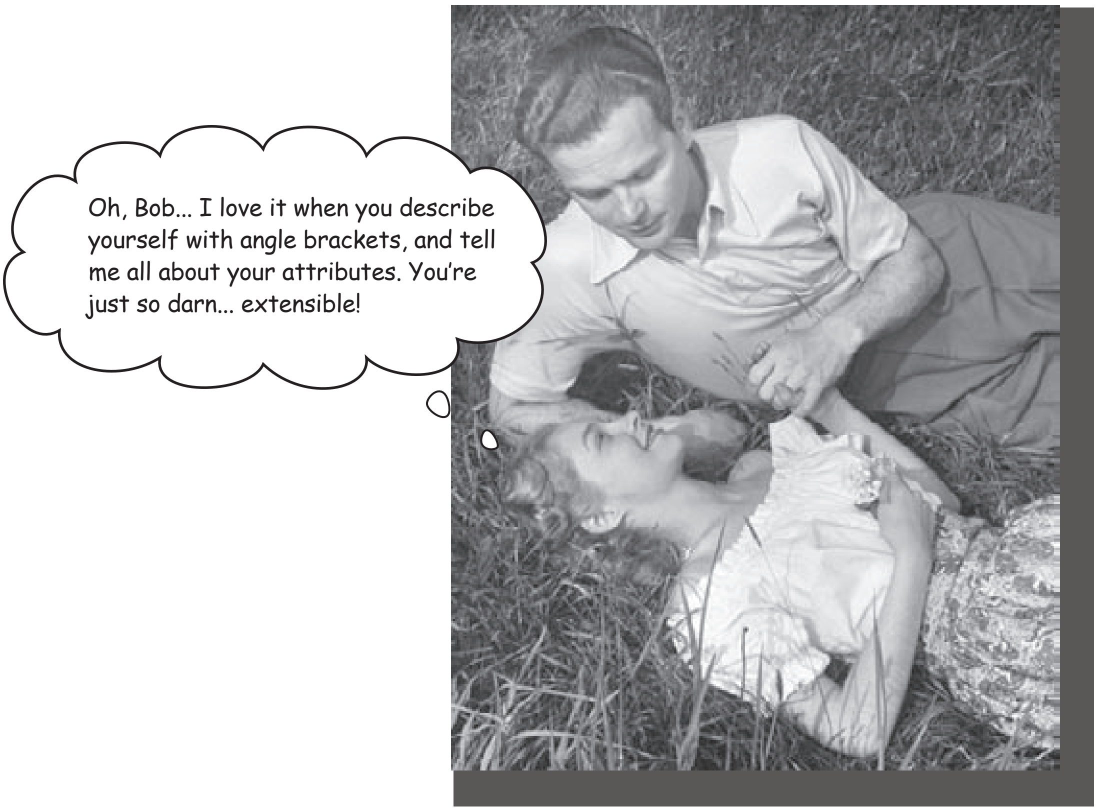
Is it important for you to really control every aspect of your code? Do you go crazy wondering how efficient every function you use in your JavaScript really is? Can you not stand the thought of missing out on learning some new tool, trick, or technique? If this is you, you may just end up frustrated and annoyed by frameworks. Stick with writing your own requests, callbacks, and utility functions, building an ever-growing library of code in `utils.js`, and not having to update to a new version of a framework every few months.

Don't care so much about every internal line of code? If you're a productivity nut, and want great apps with a minimal amount of time spent dealing with errors, weird browser inconsistencies, and oddities of the DOM, frameworks might be just for you. Say goodbye to `request.send(null)` forever, and pick a framework to learn. It shouldn't take you long... you already know what's really going on with asynchronous requests.

Either way, the choice is yours. We think it's pretty important to know what's going on under the hood, whether you use a framework or not, so the rest of this book will stick with plain old JavaScript instead of going with any particular framework. But everything you'll learn still is useful, even if you later use a framework to hide away the details of what's going on.

9 xml requests and responses

More Than Words Can Say



How will you describe yourself in 10 years? How about 20?

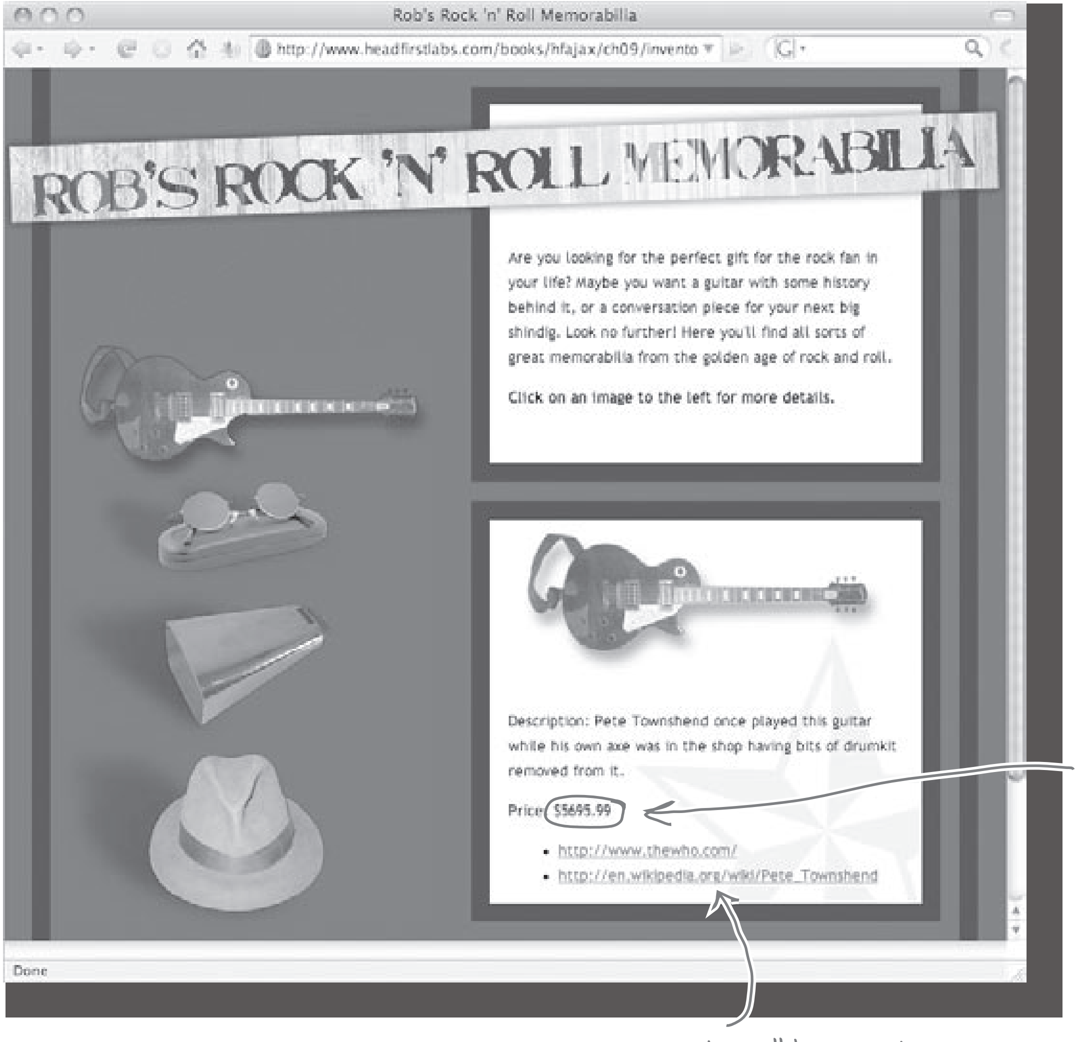
Sometimes you need **data that can change with your needs**... or the needs of your customers. Data you're using now might need to change in a few hours, or a few days, or a few months. With XML, the *extensible markup language*, your data can **describe itself**. That means your scripts won't be filled with ifs, elses, and switches. Instead, you can use the descriptions that XML provides about itself to figure out how to ***use*** the data the XML contains. The result: **more flexibility** and **easier data handling**.

As a special bonus, we're bringing back the DOM in this chapter... keep an eye out!

Classic rock gets a 21st century makeover

Rob's Rock and Roll Memorabilia has hit the big time. Since going online with the site you built for Rob, he's selling collectible gear to rich customers around the world.

In fact, Rob's gotten lots of good feedback on the site, and he's making some improvements. He wants to include a price for each item, in addition to the description, and he also wants to be able to include a list of related URLs so customers can find out more about each item.

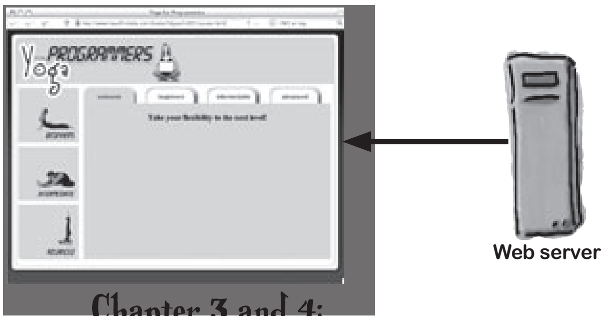




Server Response Magnets

Below are diagrams of the interactions between several of the apps you've built and programs on the server that those apps use. Can you place the right server response magnets on each diagram?

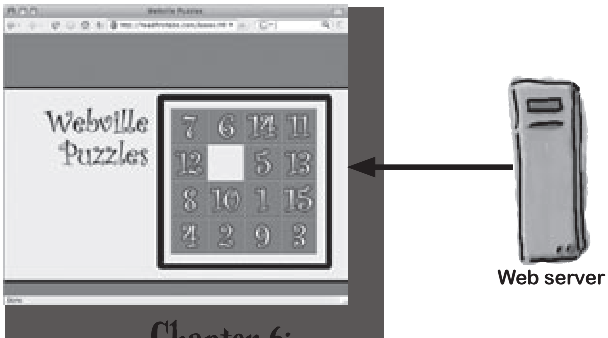
If you don't remember, flip back to these earlier chapters, or check your own code.



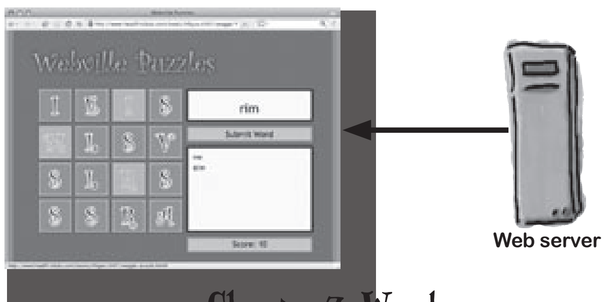
Chapter 3 and 4:
Yoga for Programmers



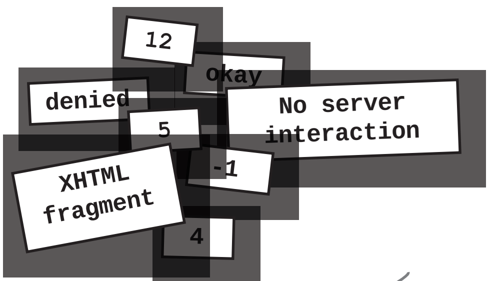
Chapter 5:
Mike's Movies



Chapter 6:
The Fifteen Puzzle



Chapter 7: Woggle



Use these magnets for the servers' responses.

Sharpen your pencil

How are the server responses from the apps you've built so far different from what Mike wants his server to respond with in the new version of his rock and roll site?

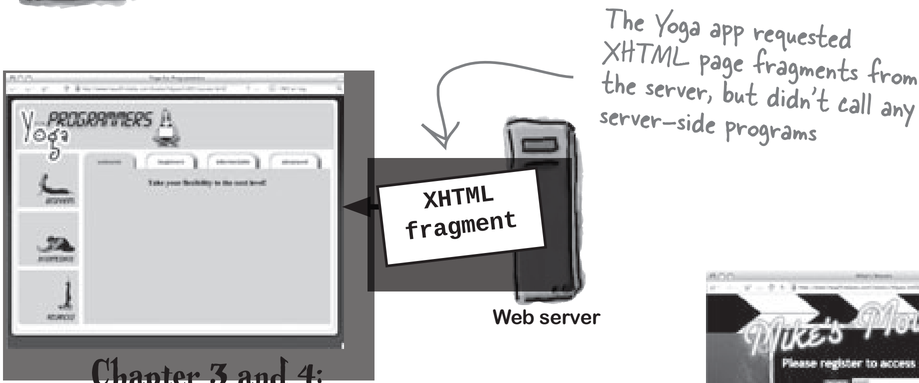
....

....

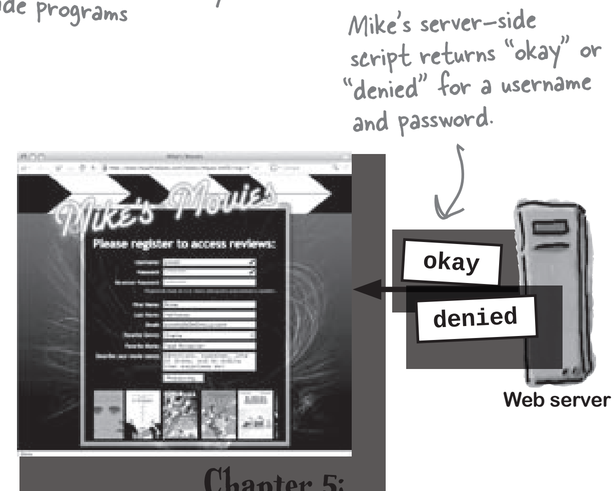


Server Response Magnet Solutions

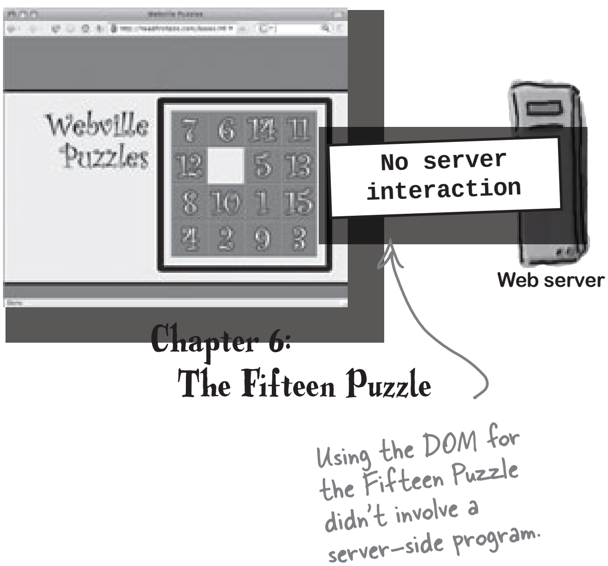
Below are diagrams of the interactions between several of the apps you've built and programs on the server that those apps use. Can you place the right server response magnets on each diagram?



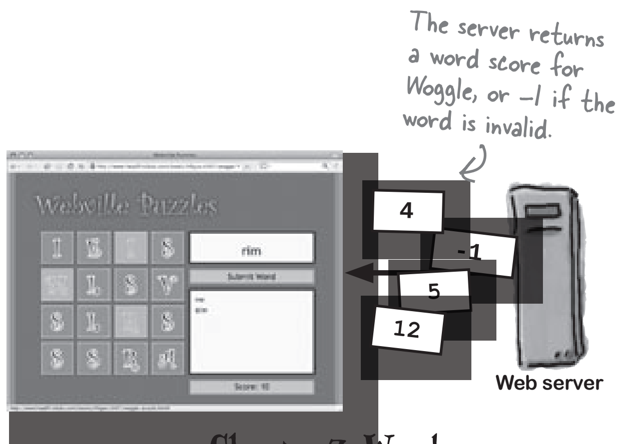
Chapter 3 and 4: Yoga for Programmers



Chapter 5: Mike's Movies



Chapter 6: The Fifteen Puzzle



Chapter 7: Woggle

Sharpen your pencil Solution

How are the server responses from the apps you've built so far different from what Mike wants his server to respond with in the new version of his rock and roll site?

All the servers so far sent out a single response... Mike's server is going to send back more than one piece of data.

How should a server send a MULTI-valued response?

So far, all the server-side programs we've worked with have sent back a single piece of data, like `-1` or "okay." But now Mike wants to send back several pieces of data at one time:

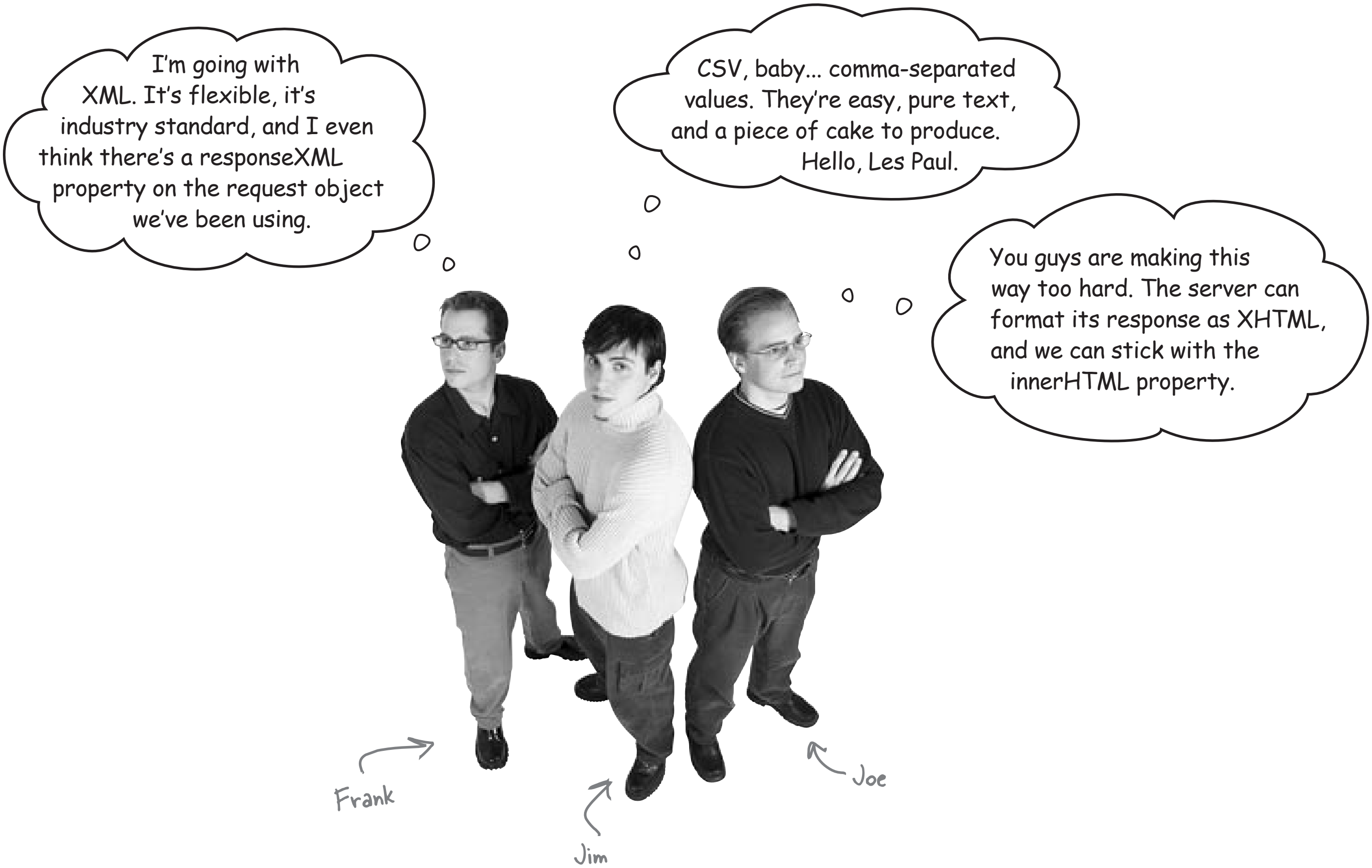
1. A string description of the selected item.
2. A numeric price for the item, like 299.95.
3. A list of URLs with related information about the item.

I tell you what... this is pretty big business these days. If you can figure out how to help me, I'll give you an all-original '59 Les Paul electric guitar, okay?

What would you do?

There are lots of ways to handle getting more than one value back from the server, and this time, the choice is up to you. What do you think?





Sharpen your pencil

Time to place your bets. We're going to follow Frank, Jim, and Joe and see which one comes up with the best solution. Who do you think will solve Rob's site problems and win the Les Paul?

- ☐ **XML is the best choice.** Frank's gonna win.
- ☐ **CSV is simple and functional.** Go, Jim!
- ☐ **innerHTML: it ain't broke.** Joe's got it in the bag.

Sharpen your pencil

You're not done sharpening that pencil just yet. Suppose you had the following information for an item:

Item ID: itemCowbell
Description: Remember the famous "more cowbell" skit from Saturday Night Live? Well this is the actual cowbell.
Price: 299.99
URLs: http://www.nbc.com/Saturday_Night_Live/
http://en.wikipedia.org/wiki/More_cowbell

How would a server represent this information...

1 **...as XML?**

In this space, write what you think the XML for this item would look like.

2 **...as CSV (comma-separated values)?**

What would the CSV look like from the server?

3 **...as an XHTML fragment?**

What about XHTML, suitable for innerHTML?

Sharpen your pencil Solution

You're not done sharpening that pencil just yet. Suppose you had the following information for an item:

Item ID: itemCowbell
Description: Remember the famous "more cowbell" skit from Saturday Night Live? Well this is the actual cowbell.
Price: 299.99
URLs: http://www.nbc.com/Saturday_Night_Live/
http://en.wikipedia.org/wiki/More_cowbell

How would a server represent this information...

1 **...as XML?**

All XML documents begin like this.

```
<?xml version="1.0"?>
<item id="itemCowbell">
 <description>Remember the famous "more cowbell" skit from
    Saturday Night Live? Well this is the actual cowbell.</description>
 <price>299.99</price>
 <resources>
   <url>http://www.nbc.com/Saturday_Night_Live/</url>
   <url>http://en.wikipedia.org/wiki/More_cowbell</url>
 </resources>
</item>
```

We used an attribute for the item ID.

The description and price are in XML elements.

We grouped the URLs with a resources element, and then put each URL in a url element.

* These solutions are just ONE way to represent the item data as XML, CSV, and XHTML. You might have come up with something a little different. As long as you got the right values in the right format, you're all set.

2 **...as CSV (comma-separated values)?**

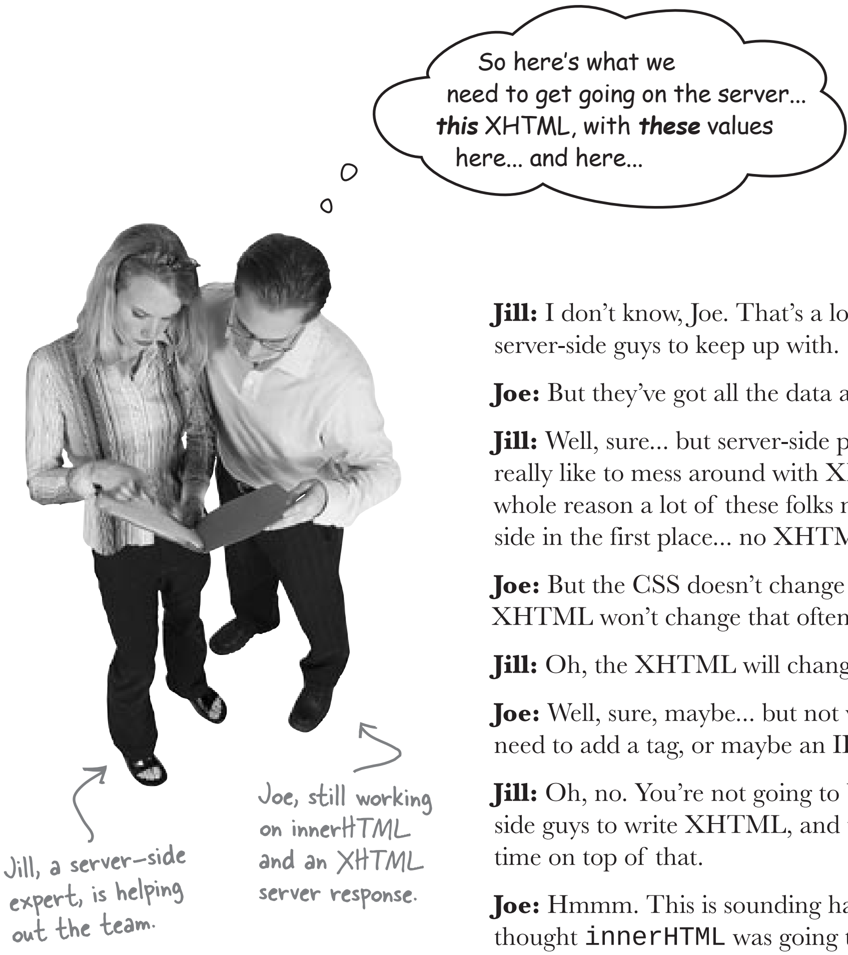
```
itemCowbell,Remember the famous 'more cowbell' skit from
    Saturday Night Live? Well this is the actual cowbell,299.99,
    http://www.nbc.com/Saturday_Night_Live/,
    http://en.wikipedia.org/wiki/More_cowbell
```

Each item in a CSV string is separated by a comma.

3 **...as an XHTML fragment?**

```
<p>Description: Remember the famous 'more cowbell" skit from
    Saturday Night Live? Well this is the actual cowbell.</p>
<p>Price: $299.99</p>
<ul>
 <li><a href="http://www.nbc.com/Saturday_Night_Live/">
        http://www.nbc.com/Saturday_Night_Live/</a></li>
 <li><a href="http://en.wikipedia.org/wiki/More_cowbell">
        http://en.wikipedia.org/wiki/More_cowbell</a></li>
</ul>
```

This is the XHTML exactly as it needs to be inserted into the rock and roll page. CSS styles it, and the data from the server is wrapped up in XHTML tags.



Jill: I don't know, Joe. That's a lot of formatting for the server-side guys to keep up with.

Joe: But they've got all the data about the item, right?

Jill: Well, sure... but server-side programmers don't really like to mess around with XHTML. That's the whole reason a lot of these folks move over to the server-side in the first place... no XHTML.

Joe: But the CSS doesn't change that much, so the XHTML won't change that often.

Jill: Oh, the XHTML will change sometimes?

Joe: Well, sure, maybe... but not very often. Only if we need to add a tag, or maybe an ID for CSS...

Jill: Oh, no. You're not going to be able to get server-side guys to write XHTML, and then change it all the time on top of that.

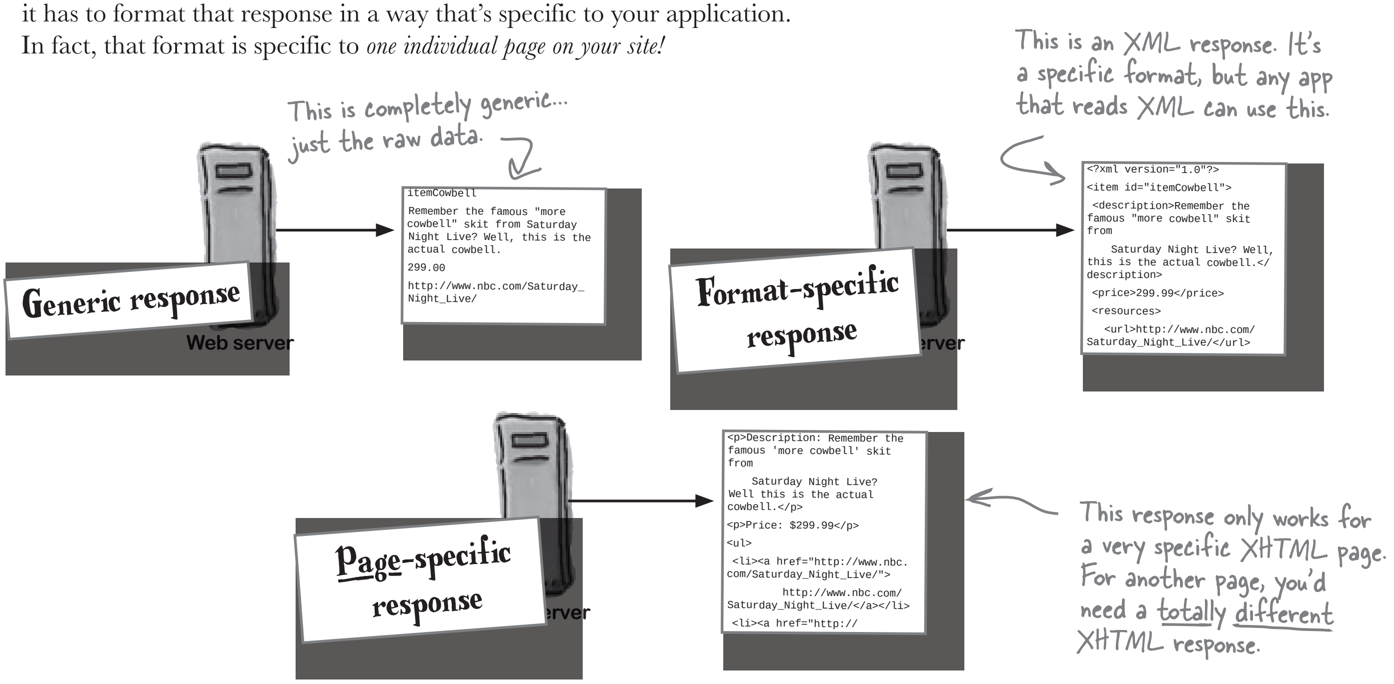
Joe: Hmmm. This is sounding harder than I thought. I thought `innerHTML` was going to be really simple...

innerHTML is only simple for the CLIENT side of a web app

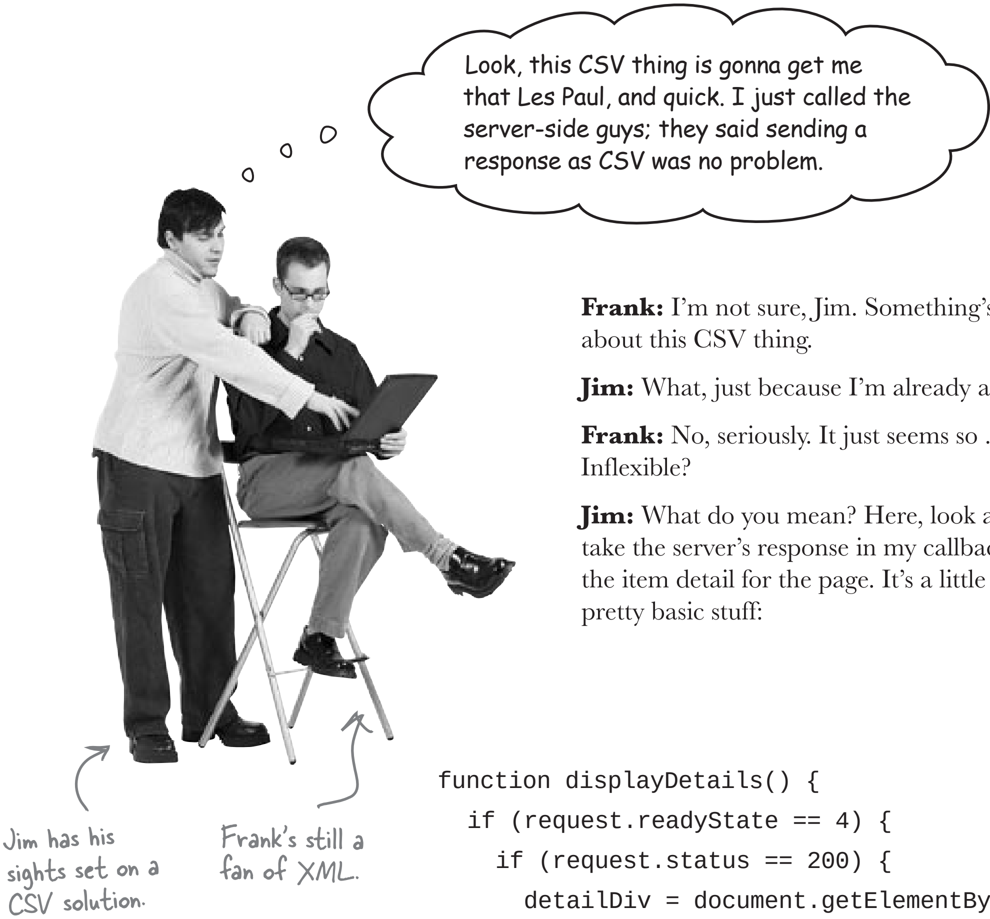
From a client-side point of view, `innerHTML` is pretty simple to use. You just get an XHTML response from the server, and drop it into a web page with an element's `innerHTML` property.

```
function displayDetails() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      detailDiv = document.getElementById("description");
      detailDiv.innerHTML = request.responseText;
    }
  }
}
```

The problem is that the server has to do a *lot* of extra work. Not only does the server have to get the right information for your app's request, it has to format that response in a way that's specific to your application. In fact, that format is specific to *one individual page on your site!*



If you were a server-side developer... which would YOU prefer?



Frank: I'm not sure, Jim. Something's bugging me about this CSV thing.

Jim: What, just because I'm already almost done?

Frank: No, seriously. It just seems so ... I don't know. Inflexible?

Jim: What do you mean? Here, look at my code to take the server's response in my callback, and update the item detail for the page. It's a little long, but it's all pretty basic stuff:

```
function displayDetails() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      detailDiv = document.getElementById("description");
      ~~detailDiv.innerHTML = request.responseText;~~

      // Remove existing item details (if any)
      for (var i=detailDiv.childNodes.length; i>0; i--) {
        detailDiv.removeChild(detailDiv.childNodes[i-1]);
      }

      // Add new item details
      var response = request.responseText;
      var itemDetails = response.split(",");
```

We no longer use innerHTML.

This code removes any elements added by previous calls to displayDetails().

First, we get the response.

Then, we separate the values using the commas.

(continued on the next page)

```
      var descriptionP = document.createElement("p");
      descriptionP.appendChild(
        document.createTextNode("Description: " +
          itemDetails[1]));
      detailDiv.appendChild(descriptionP);
      var priceP = document.createElement("p");
      priceP.appendChild(
        document.createTextNode("Price: $" + itemDetails[2]));
      detailDiv.appendChild(priceP);
      var list = document.createElement("ul");
      for (var i=3; i<itemDetails.length; i++) {
        var li = document.createElement("li");
        var a = document.createElement("a");
        a.setAttribute("href", itemDetails[i]);
        a.appendChild(document.createTextNode(itemDetails[i]));
        li.appendChild(a);
        list.appendChild(li);
      }
      detailDiv.appendChild(list);
    }
  }
}
```

This creates a new <p> with the description of the item in it.

Next, we add another <p> with the price.

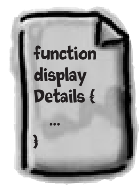
Let's display the URLs as list items in an unordered list.

Each URL goes into an <a>, which is added as the content of an <li>...

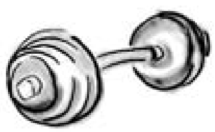
...and then the <li> gets added to a <ul>...

...which finally ends upunder the details <div>.

All of this code goes into thumbnails.js, replacing the old version of displayDetails().



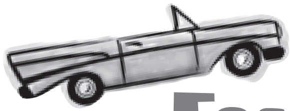
thumbnails.js



BRAIN BARBELL

Why do you think the loop deleting any pre-existing elements from the details <div> counts down, instead of counting up?

If you're stumped, try reversing the loop and see what happens. Can you figure out what's going on?



Test Drive

Try out CSV for yourself.

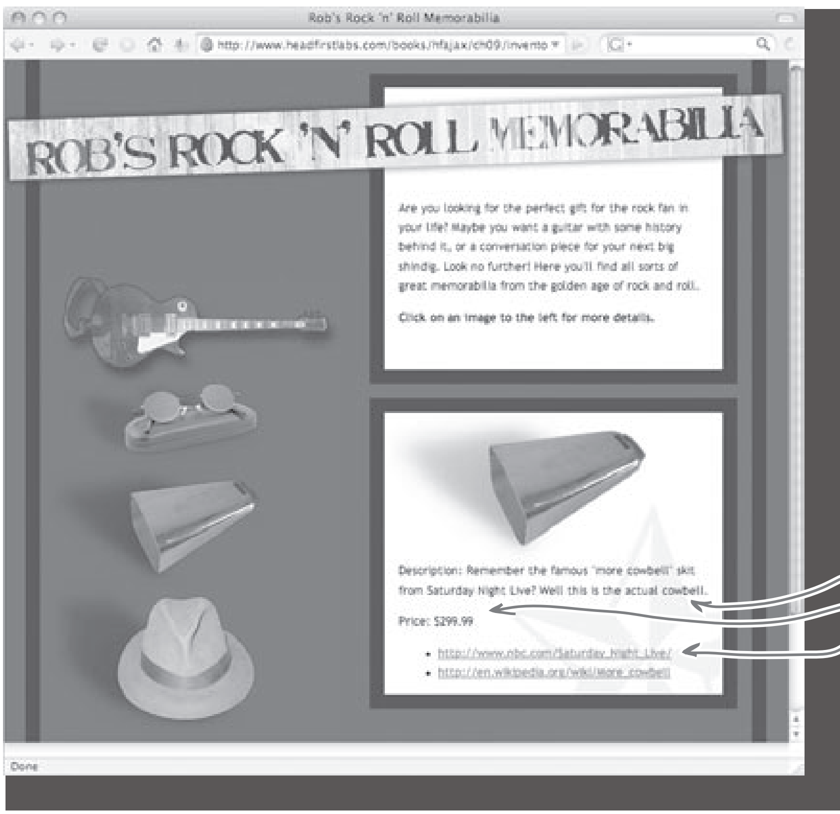
Download the examples for Chapter 9 from the Head First Labs website. Open `thumbnails.js`, and make two changes:

1. Update `displayDetails()` to match page 354.

2. In `getDetails()`, change the URL of the server-side script to `getDetailsCSV.php`.

The downloads for Chapter 9 include a server-side script that returns CSV instead of plain text.

Now try out the site. Does everything work?



This looks right... there's a description, price, and list of URLs.

there are no Dumb Questions

Q: What is CSV again?

A: CSV stands for comma-separated values. It just means that several values are put together into a single string, with commas separating each individual value.

Q: I've also heard about TSV. Is that similar?

A: TSV refers to *tab*-separated values. The idea is the same, but tabs are used instead of commas. In fact, you can use anything you want to separate the values: a pipe symbol (`|`), an asterisk (`*`), or anything else that's a fairly uncommon character.

Q: Why do you need to use an uncommon character to separate values?

A: If you use something common, like a period or letter, that same character might show up in your data. Then, your JavaScript might split the data incorrectly, giving you problems when you display or interpret that data.

In fact, CSV is a bit dangerous because an item description might have a comma in it. In that case, you'd end up splitting the description on the comma, and having all sorts of problems.

Q: So is that why we shouldn't use CSV?

A: Good question. Frank, Jim, and Joe are still debating the merits of CSV, but you could always swap out those commas for something else, and change your client code to split on that new character instead of commas. As for whether or not you should use CSV, you may want to keep reading...

Q: What is setAttribute()? I've never seen that before.

A: `setAttribute()` creates a new attribute on an element. The method takes two arguments: the name of the attribute and its value. If there's no attribute with the supplied name, a new attribute is created. If there's already an attribute with the supplied name, that attribute's value is replaced with the one you supplied to `setAttribute()`.

Q: What about childNodes? What's that?

A: `childNodes` is a property on every DOM node. The property returns an array of all the child nodes for that node. So you can get an element's children, for example, and iterate over them or delete them.

Q: So why *did* you iterate backwards over the childNodes array?

A: That's a tricky one. Here's a hint to get you thinking in the right direction: when you call `removeChild()`, the node you supply to that method is removed from its parent immediately.

That also means that all references to that now-removed node—say in an array full of an element's child nodes—have to be updated. Without a child to point to, all the child nodes that come after the removed node have to be moved up in the array.

So if you iterated over an array like `childNodes` from front to back, removing nodes as you went, what would happen?

BRAIN POWER

What do you think Frank meant on the last page about CSV being inflexible? Are there problems with the server's response being in CSV that adding new types of items might cause?

Listen, can I get an item's details in XML from the server? I'm already behind, but I still think there's a problem with that CSV solution. I want to get my XML-based version of the app working ASAP.



Jill: Sure, XML is easy. The server-side guys shouldn't have any problem with that at all. It's certainly a lot better than XHTML...

Frank: Yeah, I heard Joe was working on that. The server guys couldn't get him an XHTML response?

Jill: Well, they could've, but nobody wanted to. XHTML is a mess to work with on the server, and it changes all the time.

Frank: You know XHTML is just a flavor of XML, right?

Jill: Sure, but lots of people and apps can use XML. Dealing with a certain `<div>` with this id, or only using `<p>`'s and not `<br />`'s... that's pretty fragile.

Frank: No kidding. Well, I've got to rewrite my callback, but let me know when the XML response is ready, okay?

Frank's asked Jill for some advice from a server-side perspective.

XML is pervasive in the programming world. If you respond in XML, LOTS of different applications can work with that XML response.

there are no Dumb Questions

Q: What do you mean, "XHTML is just a flavor of XML"?

A: A flavor of XML is like a specific implementation of XML, with certain elements and attributes defined. So XHTML uses elements like `html` and `p` and `div`, and then those elements are used along with attributes and text values. You can't make up new elements, but instead you just use the ones already defined.

With XML, you can define flavors like this—sometimes called *XML vocabularies*—and extend XML for whatever your needs are. That's why XML is so flexible: it can change to match the data it represents.

You use the DOM to work with XML, just like you did with XHTML

Since XHTML is really a particular implementation of XML, it makes sense that you can use the DOM to work with XML, too. In fact, the DOM is really designed to work with XML from the ground up.

Even better, the request object you've been using to talk to the server has a property that returns a DOM tree version of the server's response. That property is called `responseXML`, and you use it like this:

```
var responseDoc = request.responseXML;
```

responseXML holds a DOM tree version of the server's response.

Sharpen your pencil

XML, XHTML... it shouldn't make much difference to an experienced DOM programmer like yourself. You've got two assignments:

- [ ] Draw a DOM tree for the XML response the server will send to Rob's app (the response is shown below).
- [ ] Write a version of the displayDetails() callback in thumbnails.js that will use the DOM to get the various parts of the server's response, and update the item's details on Rob's web page.

```
<?xml version="1.0"?>
<item id="itemCowbell">
  <description>Remember the famous "more cowbell" skit
     from Saturday Night Live? Well, this is the actual
     cowbell.</description>
  <price>299.99</price>
  <resources>
    <url>http://www.nbc.com/Saturday_Night_Live/</url>
    <url>http://en.wikipedia.org/wiki/More_cowbell</url>
  </resources>
</item>
```

This id will match the id you send the server in your request.

There will always be a single description and price element.

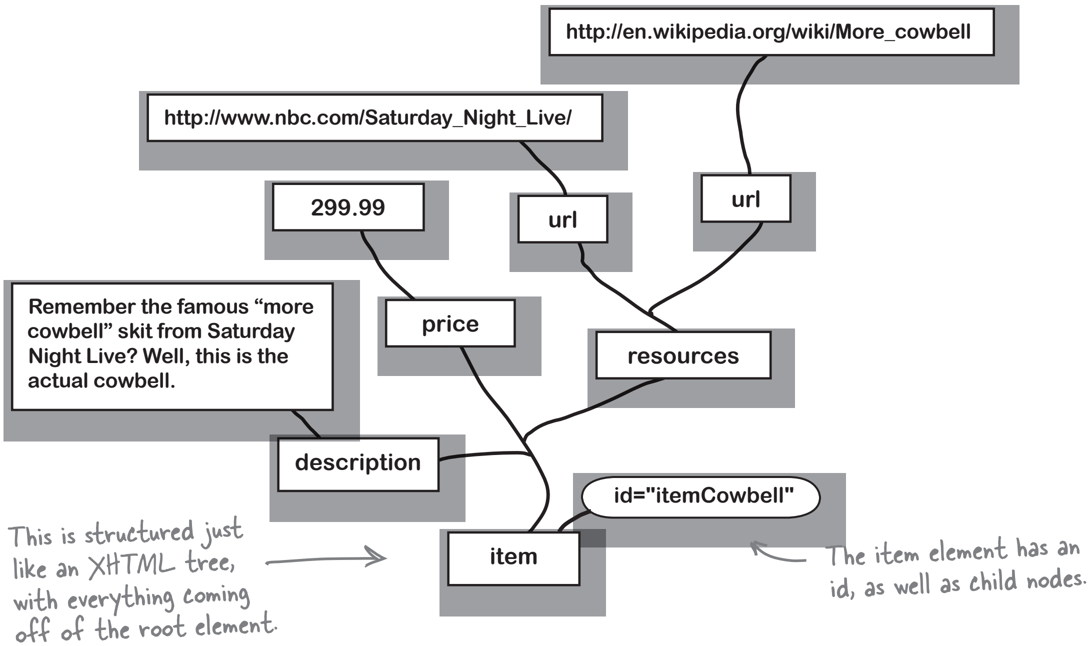
The server can send an unlimited number of URLs for each item.

Sharpen your pencil Solution

XML, XHTML... it shouldn't make much difference to an experienced DOM programmer like yourself. You had two different assignments:

☐ Draw a DOM tree for the XML response the server will send to Rob's app (the response is shown below).

```
<?xml version="1.0"?>
<item id="itemCowbell">
 <description>Remember the famous "more cowbell" skit
    from Saturday Night Live? Well, this is the actual
    cowbell.</description>
 <price>299.99</price>
 <resources>
   <url>http://www.nbc.com/Saturday_Night_Live/</url>
   <url>http://en.wikipedia.org/wiki/More_cowbell</url>
 </resources>
</item>
```



☐ Write a version of the displayDetails() callback in thumbnails.js that will use the DOM to get the various parts of the server's response, and update the item's details on Rob's web page.

Most of the code that works on the page itself is identical to the CSV version on page 354.

The big difference is in how we handle the response from the server.

First, we get the response in the form of an XML DOM tree.

We can get the <description> element, and then get its first child: a text node. From there, we just get the text node's value.

Getting the price is the same pattern: grab the element, get its text, and get that text node's value.

We can get all the <url> elements and loop through each one.

```
function displayDetails() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var detailDiv = document.getElementById("description");

      // Remove existing item details (if any)
      for (var i=detailDiv.childNodes.length; i>0; i--) {
        detailDiv.removeChild(detailDiv.childNodes[i-1]);
      }

      // Add new item details
      var responseDoc = request.responseXML;
      var description = responseDoc.getElementsByTagName("description")[0];
      var descriptionText = description.firstChild.nodeValue;
      var descriptionP = document.createElement("p");
      descriptionP.appendChild(
        document.createTextNode("Description: " + descriptionText));
      detailDiv.appendChild(descriptionP);
      var price = responseDoc.getElementsByTagName("price")[0];
      var priceText = price.firstChild.nodeValue;
      var priceP = document.createElement("p");
      priceP.appendChild(
        document.createTextNode("Price: $" + priceText));
      detailDiv.appendChild(priceP);
      var list = document.createElement("ul");
      var urlElements = responseDoc.getElementsByTagName("url");
      for (var i=0; i<urlElements.length; i++) {
        var url = urlElements[i].firstChild.nodeValue;
        var li = document.createElement("li");
        var a = document.createElement("a");
        a.setAttribute("href", url);
        a.appendChild(document.createTextNode(url));
        li.appendChild(a);
        list.appendChild(li);
      }
      detailDiv.appendChild(list);
    }
  }
}
```

Test Drive

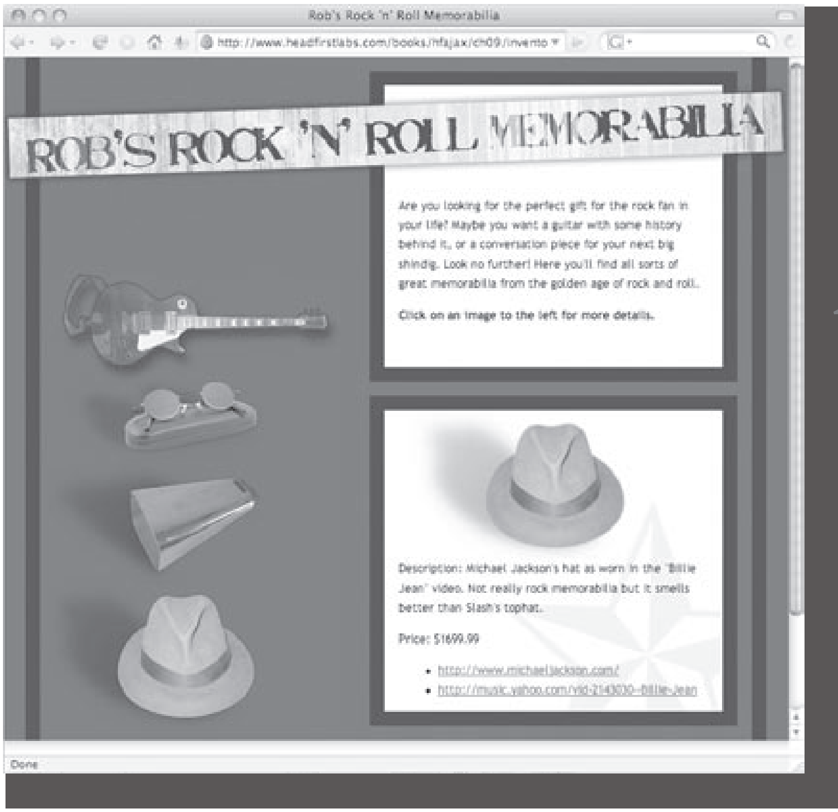
And now for an XML solution...

Open `thumbnails.js`, and make two more changes:

1. Update `displayDetails()` to match the XML version of the callback shown on page 361.

2. In `getDetails()`, change the URL of the server-side script to `getDetailsXML.php`.

How does the XML version of Rob's online shop look?



This looks just like the CSV version, but it's using XML and the DOM.

Hey guys... these are looking great. That CSV deal was quick! But I've got one more change before I declare a winner... customers want more details specific to each item. So for rock memorabilia, I might have an artist name and band name, and for instruments I might have a manufacturer and year. No big deal, right?

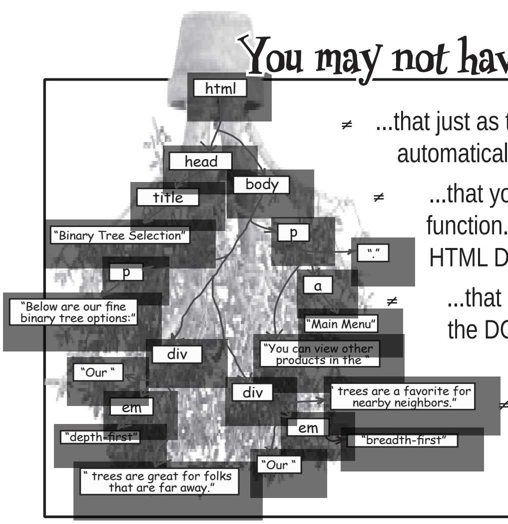


Rob wants data that changes depending on the request.

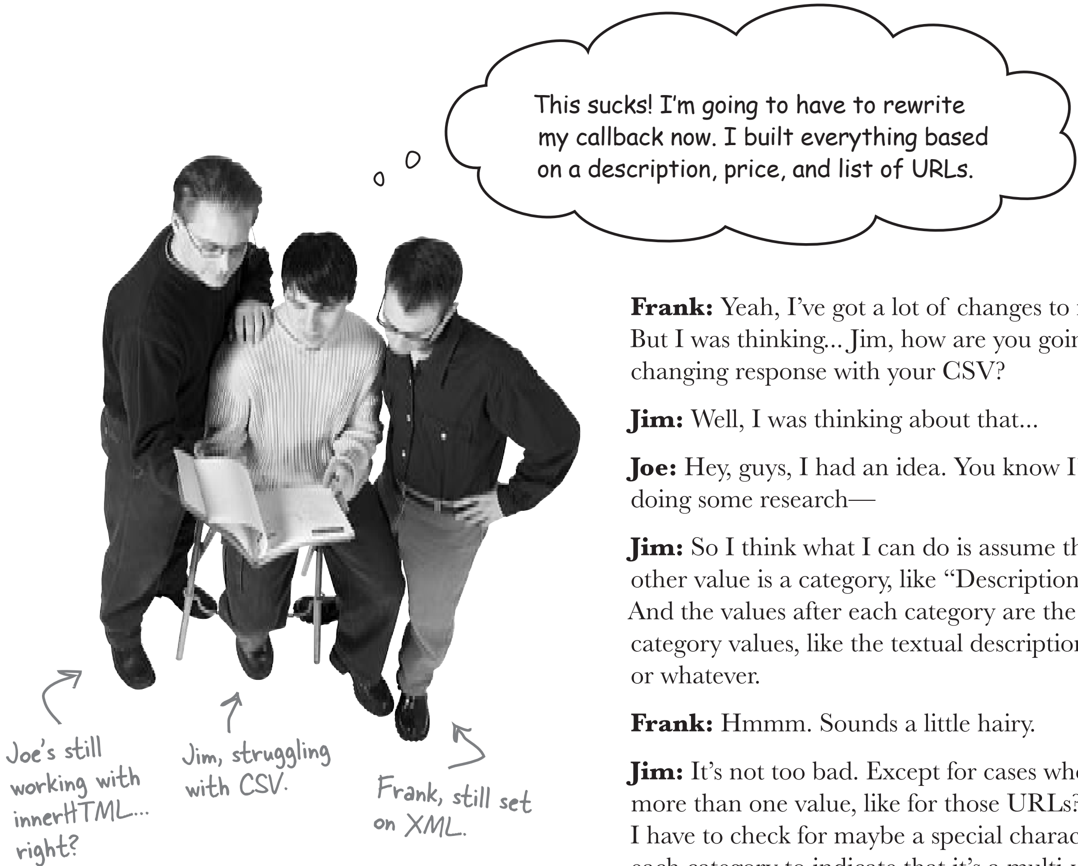
If you ask the server for details about a guitar, you'll get a manufacturer and year. Clothing? A manufacturer, sure, but also a size. And for bands, you'll get a band name, and possibly the name of the individual in the band that the item belonged to or is associated with.

How would ***you*** handle a changing response from the server? And who's better equipped to handle this new requirement? Frank, with his XML, or Jim, with his CSV?

You may not have known...



- ...that just as the browser sees your HTML as a DOM tree, web browsers automatically convert any XML they have to deal with into DOM trees.
- ...that you can work with more than one DOM tree in the same JavaScript function. For example, you can read an XML DOM tree and update an HTML DOM tree, all at the same time.
- ...that HTML elements and XML elements are both just element nodes in the DOM. There's no difference between an XML type and an HTML type, at least when it comes to the DOM.
- ...that the responseXML property always returns a DOM document object, even if that object is a single element, or just a single text node.



Frank: Yeah, I've got a lot of changes to make, too. But I was thinking... Jim, how are you going to handle a changing response with your CSV?

Jim: Well, I was thinking about that...

Joe: Hey, guys, I had an idea. You know I've been doing some research—

Jim: So I think what I can do is assume that every other value is a category, like "Description" or "Price." And the values after each category are the actual category values, like the textual description, or 399.99, or whatever.

Frank: Hmmm. Sounds a little hairy.

Jim: It's not too bad. Except for cases where there's more than one value, like for those URLs? Then I think I have to check for maybe a special character before each category to indicate that it's a multi-value category.

Joe: Listen, guys, I wanted to show you—

Frank: Wow, Jim, that's nasty. Sounds like this latest change from Rob is really going to be a pain.

Jim: Yeah, it kinda is. But what else can I do?

You can't always know in advance what the data structure you get from the server will look like.

And even if you do, that format might change... at anytime.


Wouldn't it be dreamy if there was actually a data format that described itself? It could tell me what every bit of data was used for. But I suppose it's just a fantasy...

XML is self-describing

The thing that's cool about XML is that you can create your own vocabulary. XHTML is an XML vocabulary that's specific to displaying things on the web. But suppose we needed a vocabulary for describing items, like at Rob's online store.

But the format can't be locked into elements like `<price>` or `<resources>` because we want each item to define its own categories. We might use something like this:

```
<?xml version="1.0"?>
<item id="item ID">
  <category>
    <name>Label for this category</name>
    <value>The value to display for this category</value>
  </category>
  <category>
    <name>Name of the next category</name>
    <value>Next value</value>
  </category>
  <category type="list">
    <name>Name of multi-valued category</name>
    <value>First value for this category</value>
    <value>Second value for this category</value>
  </category>
  ...
</item>
```

<item> is the root element. It's the container for all the <category> elements, just like the <html> element in an XHTML file.

The <category> element contains the label and value for each bit of information we need to display.

Every category has a <name> and a <value>. They contain the actual data we'll display.

We can have multi-valued categories, and even indicate that with an attribute on the <category> element.

The XML can contain as many <category> elements as necessary. We don't need to know how many there are or what they are in advance.

Sharpen your pencil

Here's some more data from Rob's inventory database:

Item ID: itemGuitar
Manufacturer: Gibson
Model: Les Paul Standard
Description: Pete Townshend once played this guitar while his own axe was in the shop having bits of drumkit removed from it.
Price: 5695.99
URLs: http://www.thewho.com/
http://en.wikipedia.org/wiki/Pete_Townshend

How would you represent this item's details using the XML format from the last page?

Write your XML in right here.

Sharpen your pencil Solution

Here's some more data from Rob's inventory database:

Item ID: itemGuitar
Manufacturer: Gibson
Model: Les Paul Standard
Description: Pete Townshend once played this guitar while his own axe was in the shop having bits of drumkit removed from it.
Price: 5695.99
URLs: http://www.thewho.com/
http://en.wikipedia.org/wiki/Pete_Townshend

Your job was to represent this in XML using the vocabulary from page 366.

```
<?xml version="1.0"?>
<item id="itemGuitar">
  <category>
    <name>Manufacturer</name>
    <value>Gibson</value>
  </category>
  <category>
    <name>Model</name>
    <value>Les Paul Standard</value>
  </category>
  <category>
    <name>Description</name>
    <value>Pete Townshend once played this guitar while his own axe
         was in the shop having bits of drumkit removed from it.</value>
  </category>
  <category>
    <name>Price</name>
    <value>5695.99</value>
  </category>
  <category type="list">
    <name>URLs</name>
    <value>http://www.thewho.com/</value>
    <value>http://en.wikipedia.org/wiki/Pete_Townshend</value>
  </category>
</item>
```

Most of this is just "fill in the blanks." You drop in the name of a category and its value, and you're all set.

The URLs are a list, so we have to set the category type to "list."

Exercise

It's time for the big finish (at least for now). Your job is to take what you've learned about the DOM, server-side responses in XML, and the format from the last few pages, and put it all together. Here's what you've got to do:

- [ ] Change your request URL to use getDetailsXML-updated.php. That script is in with the other downloads for this chapter from Head First Labs.

- [ ] Rewrite the displayDetails() callback to work with the XML vocabulary we've been looking at. Remember, you may get more—or less—categories for different items. And you've got to handle those list categories, too.

- [ ] Test everything out! Once you've got everything working, turn the page to claim your Les Paul (at least, that's what we hope)!

there are no Dumb Questions

Q: So the big deal about XML is that it describes itself? That can't be useful all that often...

A: Actually, self-describing data is useful in a number of situations, just like here, with Rob's online store. It's pretty convenient to be able to define elements and structure that's suited to your business.

Even better, XML is a standard, so tons of people know how to work with it. That means your vocabulary is usable by lots of programmers, in client-side and server-side programs.

Q: Wouldn't it be easier to just make up our own data format?

A: It might seem that way at first, but proprietary data formats—ones that you make up for your own use—can really cause a lot of problems. If you don't document them, people may forget how they work. And if anything changes, you need to make sure everything is up-to-date: the client, the server, the database, the documentation… that can be a real headache.

Q: Okay, I get why we should use XML, but doesn't it become a "proprietary data format" when we start declaring element names?

A: No, not at all. That's the beauty of XML: it's flexible. The server and the client need to be looking for the same element names, but you can often work that out at run-time. That's what's meant by *self-describing*: XML describes itself with its element names and structure.

Exercise Solution

It's time for the big finish (at least for now). Your job was to take what you've learned about the DOM, server-side responses in XML, and the format from the last few pages, and complete an updated version of the displayDetails() callback.

```
function displayDetails() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var detailDiv = document.getElementById("description");

      // Remove existing item details (if any)
      for (var i=detailDiv.childNodes.length; i>0; i--) {
        detailDiv.removeChild(detailDiv.childNodes[i-1]);
      }



      // Add new item details
      var responseDoc = request.responseXML;
      var categories = responseDoc.getElementsByTagName("category");
      for (var i=0; i<categories.length; i++) {
        var category = categories[i];
        var nameElement = category.getElementsByTagName("name")[0];
        var categoryName = nameElement.firstChild.nodeValue;
        var categoryType = category.getAttribute("type");
        if ((categoryType == null) || (categoryType != "list")) {
          var valueElement = category.getElementsByTagName("value")[0];
          var categoryValue = valueElement.firstChild.nodeValue;
          var p = document.createElement("p");
          var text = document.createTextNode(
            categoryName + ": " + categoryValue);
```

This is the same as before. We start by getting rid of any existing content.

First up, we get the categories...

...and then get the name and type of each category.

This gets categories with no type attribute, or a type with a value other than "list."

We can check the type to see if it's a list. If not...

...get the value, create a <p>, and add text with the category name and value.

```
    p.appendChild(text);
    detailDiv.appendChild(p);
  } else {
    var p = document.createElement("p");
    p.appendChild(document.createTextNode(categoryName));
    var list = document.createElement("ul");
    var values = category.getElementsByTagName("value");
    for (var j=0; j<values.length; j++) {
      var li = document.createElement("li");
      li.appendChild(
        document.createTextNode(values[j].firstChild.nodeValue));
      list.appendChild(li);
    }
    detailDiv.appendChild(p);
    detailDiv.appendChild(list);
  }
}
```

This block handles lists of values.

First, we get all the values.

For each value, we add an <li> to an unordered list (<ul>).

Add both the list heading and the list itself to the <div>.

Check it out... this is even better than my old version! I can handle anything now, including those multi-valued categories. Hey, how's your CSV coming along?



<mumble, mumble>

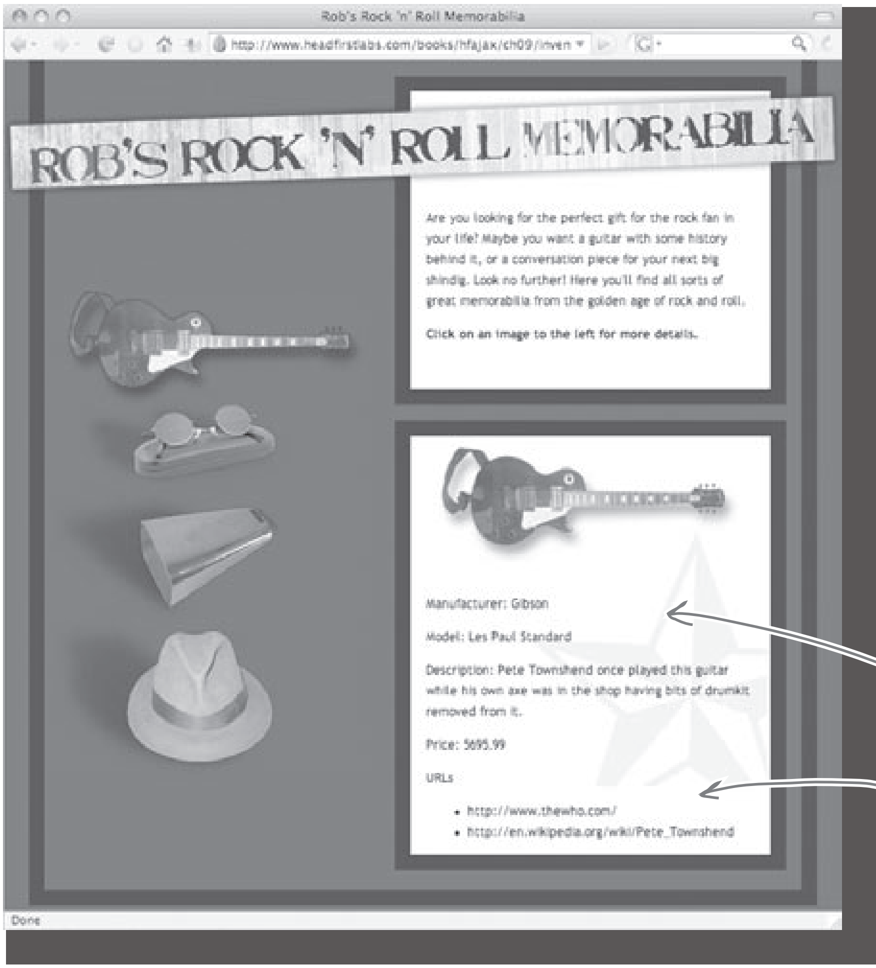
Things don't look like they're going so well for Jim and his CSV solution.

Test Drive

Test out the new, improved, more flexible version of Rob's page.

Once you've got all your code updated, let's take it for a spin, and show Rob what we've come up with. Here's what the page looks like now:

Be sure to change the URL in your getDetails() request function, too.

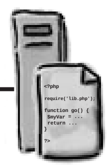


Beautiful... no matter what the XML sends, our page displays it.

Multi-valued data works, too.

Which Data Format?

Welcome to this week's edition of "Which Data Format?" You've got to decide which data format is best for the 5 examples below. Be careful: some are requests, while others are responses. Good luck!

	Text	or	XML
← Top 10 iTunes downloads of 2007 	✸		✸
 Request today's house blend →	✸		✸
 Update journal with new entry →	✸		✸
← Number of hobbits that fit in a Volkswagen 	✸		✸
 Play "When It Falls" next →	✸		✸

Answers on page 377.



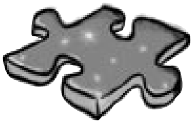


What's Joe talking about?

What ever happened to innerHTML?

And how does he plan to top XML?

Find the answers to these questions and more... all in Chapter 10.



XMLAcrostic

Take some time to sit back and give your right brain something to do. Answer the questions in the top, then use the letters to fill in the secret message.

Our original version put pre-formatted XHTML in this property:

___ ___ ___ ___ ___ ___ ___ ___ ___
1 2 3 4 5 6 7 8 9

We addded a
 element to the detailDiv to do this:

___ ___ ___ ___ ___ ___
10 12 13 14 15 16

This property of the request object contains text returned by the server:

___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
17 18 19 20 21 22 23 24 25 26 27 28

The response to our request is generated here:

___ ___ ___ ___ ___ ___   ___ ___ ___ ___
29 30 31 32 33 34   35 36 37 38

The browser puts the XML DOM into a property of this object:

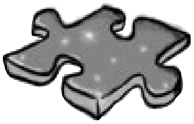
___ ___ ___ ___ ___ ___ ___
39 40 41 42 43 44 45

This was our client in this chapter:

___ ___ ___
46 47 48

___ ___ ___   ___ ___   ___ ___ ___ ___ ___ ___ ___
27 8 9   1 44   32 4 39 48 21 35 4

___ ___ ___   ___ ___ ___ ___ ___ ___ ___ ___
48 42 28   10 9 40 27 36 48 9 30



XMLAcrostic

Take some time to sit back and give your right brain something to do. Answer the questions in the top, then use the letters to fill in the secret message.

Our original version put pre-formatted XHTML in this property:

I	N	N	E	R	H	T	M	L
1	2	3	4	5	6	7	8	9

We addded a
 element to the detailDiv to do this:

F	O	R	M	A	T
10	12	13	14	15	16

This property of the request object contains text returned by the server:

R	E	S	P	O	N	S	E	T	E	X	T
17	18	19	20	21	22	23	24	25	26	27	28

The response to our request is generated here:

S	E	R	V	E	R	-	S	I	D	E
29	30	31	32	33	34		35	36	37	38

The browser puts the XML DOM into a property of this object:

R	E	Q	U	E	S	T
39	40	41	42	43	44	45

This was our client in this chapter:

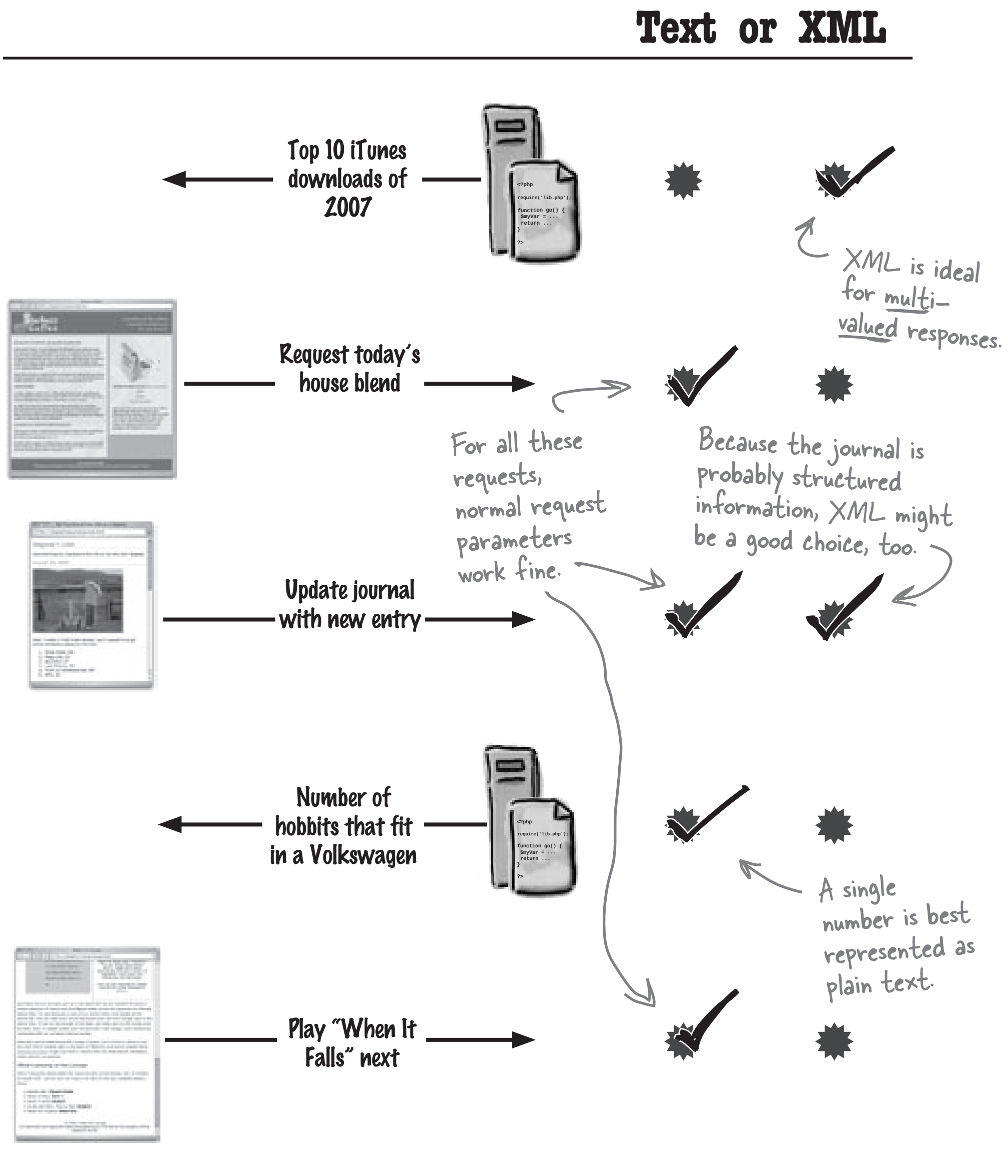
R	O	B
46	47	48

X	M	L		I	S		V	E	R	B	O	S	E
27	8	9		1	44		32	4	39	48	21	35	4

B	U	T		F	L	E	X	I	B	L	E
48	42	28		10	9	40	27	36	48	9	30

Which Data Format?

Welcome to this week's edition of "Which Data Format?" You've got to decide which data format is best for the 5 examples below. Be careful: some are requests, while others are responses. Good luck!



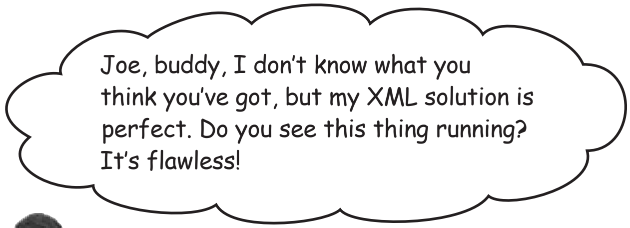
10 json

SON of JavaScript



JavaScript, objects, and notation, oh my!

If you ever need to represent objects in your JavaScript, then you're going to love JSON, **JavaScript Standard Object Notation**. With JSON, you can **represent complex objects and mappings** with text and a few curly braces. Even better, you can **send and receive JSON** from other languages, like PHP, C#, Python, and Ruby. But how does JSON stack up as a data format? Turn the page and see...





Joe: I think we might have different definitions of flawless, man. ***Two*** DOM trees to work with, and dealing with whitespace in the server's response?

Jim: That's a good point, Frank. Did you even check for whitespace nodes?

Frank: No, but that's easy enough to add in—

Joe: And your code will get even more convoluted.

Frank: Hey, at least my code works. And that CSV stuff was a total bust.

Jim: It worked great! At least... well, it worked pretty well until we had data where the structure changed depending on the item.

Joe: So you've got broken CSV, or convoluted XML. What a choice! Good thing there's ***another*** option.

Jim: What? What did you find?

Frank: This better not be another innerHTML fiasco...

Joe: I found JSON!

Jim and Frank: JSON? What the heck is that?

Joe: JSON is JavaScript Standard Object Notation. It's a way to represent a JavaScript object in plain text. So the server can send us JSON—which is just text, no XML or DOM issues to deal with—and our JavaScript can work with that response as an object.

Jim: What's the big deal about that?

Frank: Hmm. Well, if Joe's really onto something—

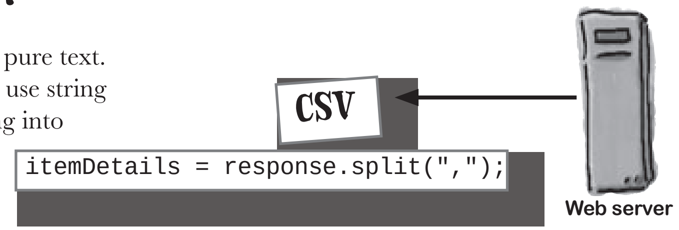
Joe: I am!

Frank: —then you wouldn't need all this DOM stuff, or even `split()` and other text manipulation code. You could just say, for example, `var description = itemDetails.description`. That ***would*** be pretty cool.

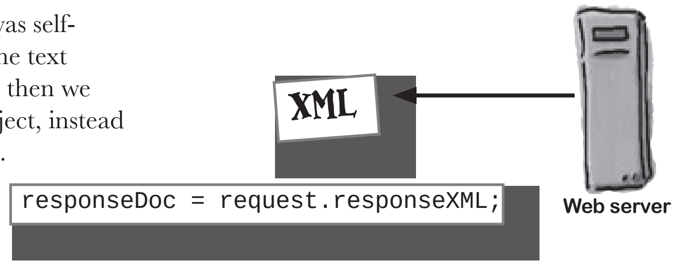
Joe: Look, here's how it works...

JSON can be text AND an object

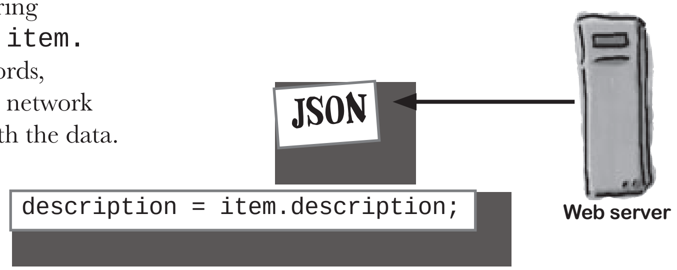
With CSV, comma-separated values, the CSV data was pure text. The server sent over the text, and our JavaScript had to use string manipulation routines like, `split()`, to turn the string into individual pieces of data.



With XML, the server sent text over, too, but that text was self-describing. So we could get a DOM representation of the text using the request object's `responseXML` property. But then we had to use all those DOM methods to work with the object, instead of actual property names like `description` or `urls`.



But suppose we had a way to get text from the server, and then treat that text as a JavaScript object. Instead of using string manipulation or DOM methods, we'd just use code like `item.description` or `itemDetails.urls`. In other words, we'd have a format that was represented as text for easy network transmission, but an object when we needed to work with the data.



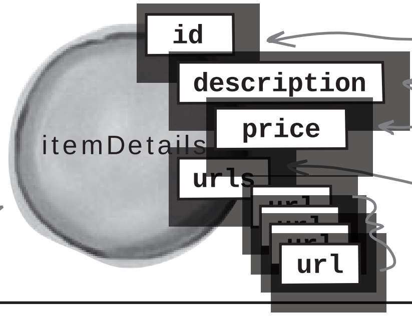
Sharpen your pencil

Suppose you had an item's ID, description, price, and a list of URLs for that item. What do you think an object representing that information might look like? Draw a circle for the object below, and then add in whatever fields you think the object might have.

Sharpen your pencil Solution

Suppose you had an item's ID, description, price, and a list of URLs for that item. Your job was to draw what you thought an object representing that information might look like.

We figured the object might best be called itemDetails, or something similar. It doesn't represent an item, but information about the item.



id, description, and price are just properties of the object.

urls is a list, where each item in the list is an individual URL.

JSON data can be treated as a JavaScript object

Don't worry... we're going to talk about how to get the JSON data from the server in a minute.

When you get JSON data from a server or some other source, you're getting text... but that text can easily be turned into a JavaScript object. Then, all you need to do is access that object's fields using **dot notation**. Dot notation just means you put the object name, and then a dot, and then a field name, like this:

```
var weakness = superman.weakness;
```

For instance, suppose you had an object like the one shown in the solution above. How do you think you'd access the value of the description field?

....

If you're looking at your answer, and thinking it's too simple, then you've probably got things just right. Working with JavaScript objects is about as easy as it gets.



So how do we get JSON data from the server's response?

When a server sends its response as JSON data, that data comes across the network as text. So you can get that data using the `responseText` property of your request object:

```
var jsonData = request.responseText;
```



Let's see exactly what the server responds with... then, we can figure out how to turn that response into something we can work with.

- [ ] Download the examples for Chapter 10, which include a JSON-specific version of the server-side script, as well as a JSON library that script uses.
- [ ] Change the request URL in getDetails() (in thumbnails.js) to point to the JSON-specific script, getDetailsJSON.php. Everything else about the request should stay the same.
- [ ] Get the textual response from the server, and display it using an alert() or some other JavaScript output function. What does the response look like?

Exercise Solution

What did the server respond with? Does it look like a JavaScript object?

- [ ] Download the examples for Chapter 10, which include a JSON-specific version of the server-side script, as well as a JSON library that script uses.

- [ ] Change the request URL in getDetails() (in thumbnails.js) to point to the JSON-specific script, getDetailsJSON.php. Everything else about the request should stay the same.

```
var url= "getDetailsJSON.php?ImageID=" + escape(itemName);
```

This is the line we used from getDetails() that requests a response from the JSON server-side script.

- [ ] Get the textual response from the server, and display it using an alert() or some other JavaScript output function. What does the response look like?

```
alert(request.responseText);
```

We added this line into the displayDetails() callback.

Here's what we got reloading the inventory page, and clicking on the guitar image.



What the heck is this? And what do we <u>DO</u> with it?

JavaScript can evaluate textual data

JavaScript is pretty good at turning text into objects, functions, and lots of other things. You can give JavaScript some text, and it's smart enough to figure out what that text represents.

For example, remember how we've been assigning event handlers?

```
image.onclick = function () {
      var detailURL = 'images/' + this.title + '-detail.jpg';
      document.getElementById("itemDetail").src = detailURL;
      getDetails(this.title);
}
```

It looks like we're assigning a textual description of a function to the image's onclick event.

JavaScript takes this textual function, and creates an actual function in memory. So when an image is clicked, the function code sitting in memory is executed. That all happens behind the scenes, though, and isn't something you need to worry about.

But what about when you have text, and you need to TELL JavaScript to turn it into something more than text?

```
{"id":"itemGuitar",
 "description":"Pete Townshend once played this guitar ...",
 "price":5695.99,
 "urls":["http://www.thewho.com/",
        "http://en.wikipedia.org/wiki/Pete_Townshend"]}
```

This response from the server looks like it's a bunch of property names and values... but how can we tell JavaScript to turn this into something we can use?

Use eval() to manually evaluate text

The `eval()` function tells JavaScript to actually evaluate text. So if you passed the text describing a statement to `eval()`, JavaScript would actually run that statement, and give you the result:

```
alert(eval("2 + 2"));
```

→ **4**

JavaScript evaluates the text "2 + 2"...

...and turns it into the statement 2 + 2...

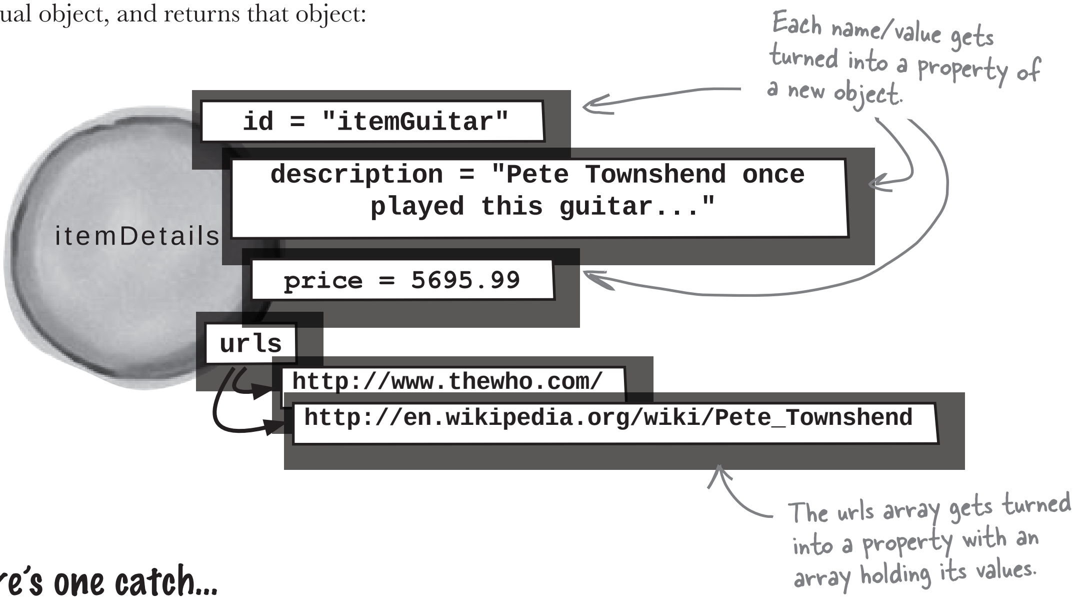
...and evaluates that expression, returning the result.

Evaluating JSON data returns an object representation of that data

So how does this apply to JSON data? Well, when you run `eval()` on text that describes a set of property names and values, then JavaScript returns an object representation of those properties and values. Suppose you ran `eval()` on this text:

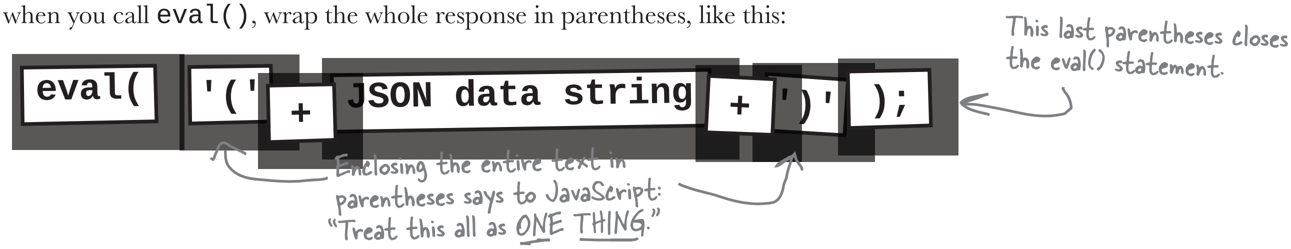
```
{"id":"itemGuitar",
 "description":"Pete Townshend once played this guitar ...",
 "price":5695.99,
 "urls":["http://www.thewho.com/",
        "http://en.wikipedia.org/wiki/Pete_Townshend"]}
```

JavaScript figures, "Hey, this looks like an object." So it turns this data into an actual object, and returns that object:



But there's one catch...

It looks like the object JavaScript creates from a JSON response is perfect for Rob's rock inventory page. There's just one thing to watch out for. You need to make sure that the overall JSON response string is seen as a single object. So when you call `eval()`, wrap the whole response in parentheses, like this:



there are no Dumb Questions

Q: Do I need any special libraries to read JSON data?

A: No. `eval()` is built into JavaScript, and it's all you need to turn JSON data into a JavaScript object.

Q: Why should I mess with eval()? Couldn't I just parse the raw text from the server?

A: You could, but why bother? `eval()` turns all that text into a very simple object, and you can avoid counting characters and messing with `split()`.

Q: eval() just stands for evaluate, right?

A: Right. `eval()` *evaluates* a string.

Q: So eval() runs a piece of text?

A: Well, not always. `eval()` takes a string, and turns it into an expression. Then, the result of that expression is returned. So for a string like "2 + 2", the expression would be `2 + 2`, and the result of that expression is `4`. So `4` is returned from `eval("2 + 2");`

But take a string like '{"id":"itemGuitar","price":5695.99"}.' Turning that into an expression and executing the expression results in a new object, not a specific "answer." So sometimes `eval()` doesn't really run text as much as it *evaluates* (or *interprets*) text.

Q: What are those curly braces around everything in the server's response?

A: JSON data is enclosed within curly braces: `{` and `}`. It's sort of like how an array is enclosed within `[` and `]`. It's just a way of telling JavaScript, "Hey, I'm about to describe an object."

Q: And each name/value pair in the text becomes a property of the object and a value for that property?

A: Right. The text "id":"itemGuitar" in an object description tells JavaScript that there's an **`id`** property, and the value of that property should be "itemGuitar."

Q: What about the urls property? That looks sort of weird.

A: urls is an array. So the property name is "urls," and the value is an array, indicated by those opening and closing square brackets (`[` and `]`).

Exercise

You've got a script that returns JSON data, and now you know how to convert that response into an object. All that's left to do is use that object in your callback. Open up thumbnails.js, and see if you can rewrite displayDetails() to convert the JSON data from the server into an object, and then use that object to update Rob's inventory page.



You've got a script that returns JSON data, and now you know how to convert that response into an object. Your job was to use that object in your callback. How did you do?

```
function displayDetails() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var detailDiv = document.getElementById("description");

      var itemDetails = eval('(' + request.responseText + ')');

      // Remove existing item details (if any)
      var children = detailDiv.childNodes;
      for (var i=children.length; i>0; i--) {
        detailDiv.removeChild(children[i-1]);
      }

      // Add new item details
      var descriptionP = document.createElement("p");
      descriptionP.appendChild(
        document.createTextNode("Description: " + itemDetails.description));
      detailDiv.appendChild(descriptionP);
      var priceP = document.createElement("p");
      priceP.appendChild(
        document.createTextNode("Price: $" + itemDetails.price));
      detailDiv.appendChild(priceP);
      var list = document.createElement("ul");
      for (var i=0; i<itemDetails.urls.length; i++) {
        var url = itemDetails.urls[i];
        var li = document.createElement("li");
        var a = document.createElement("a");
        a.setAttribute("href", url);
        a.appendChild(document.createTextNode(url));
        li.appendChild(a);
        list.appendChild(li);
      }
      detailDiv.appendChild(list);
    }
  }
}
```

Here's where we ask JavaScript to convert the server's response into an object.

Remember these extra parentheses!

There's no need for using DOM to get values from the server with JSON.

Most of this code to update the display of the page itself is identical to the XML version from Chapter 9.

Getting the details about each item is really simple now.

This code is a bit shorter than the XML version, and only uses one DOM. Do you think this version is better or worse than the XML version?

there are no Dumb Questions

Q: This is all just about a data format, right?

A: That's right. Any time you send information between your web page and a server, you're going to need some way to format that information. So far, we've used plain text to send requests, and text and XML to retrieve responses. JSON is just one more way to send data back and forth.

Q: If we've already got XML and text as options, why do we need JSON?

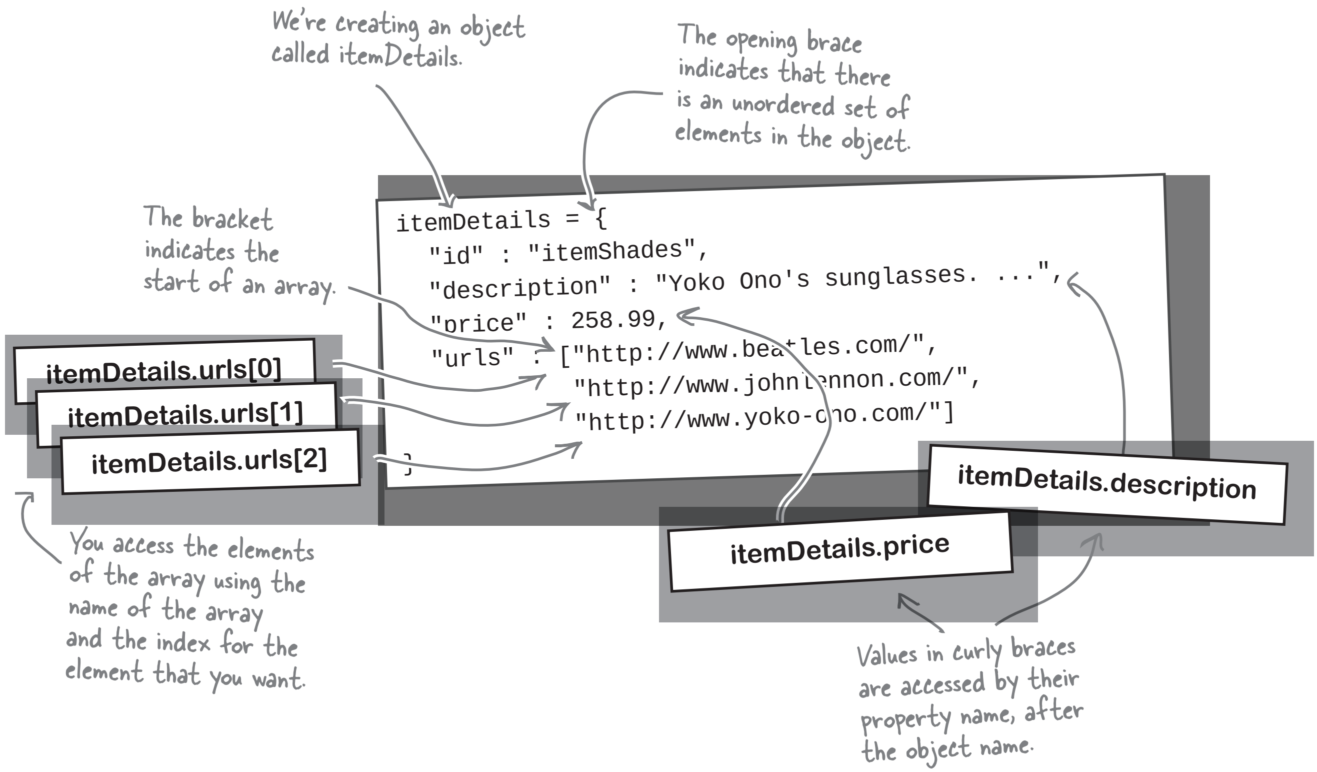
A: Since JSON is JavaScript, it's often a lot easier for both JavaScript programmers and browsers to work with. Also, because JSON creates a standard JavaScript object, it winds up looking more like a "business object" that combines data and functionality, instead of an untyped XML DOM tree. You can create similar objects from an XML response, but it requires a lot of additional work, with schemas and databinding tools.

Q: So JSON does things XML can't do?

A: It's not so much that it does more; JSON actually does *fewer* things than XML. But what JSON does, it does simply and elegantly, without a lot of the overhead that XML requires to do all the bazillion things a more fully-featured markup language was designed to handle.

Q: Can we go back to syntax? I'm still a little fuzzy on the textual representation of an item. Can you explain how that's working again?

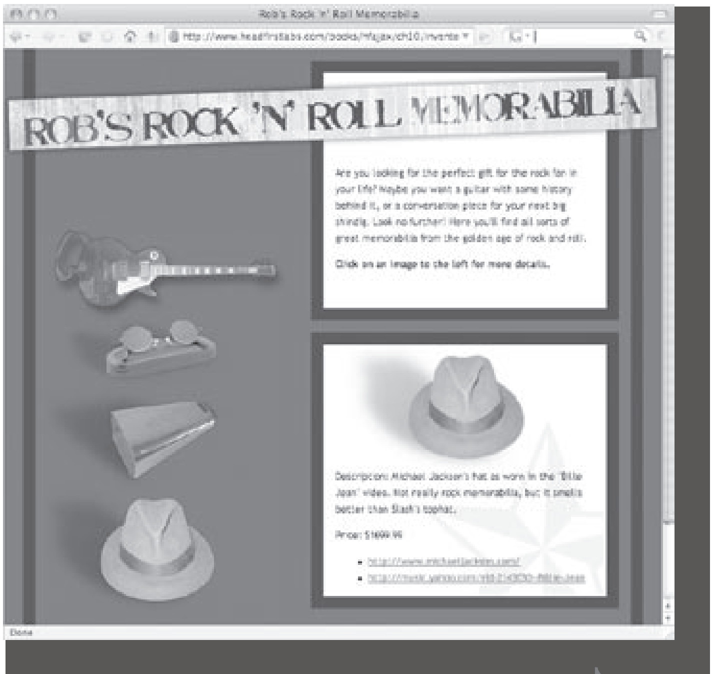
A: The curly braces, `{` and `}`, define an object, which is an unordered set of name/value pairs. Square brackets, `[` and `]`, indicate an *ordered* array. In your code, you reference elements inside curly braces by their name, but the ones inside square brackets are referenced by number. Here's a closer look:



Test Drive

But does JSON actually impress Rob?

The code looks a bit simpler, and there's one less DOM to work with. But does the JSON version of Rob's inventory page actually work? Change your callback to match the version on page 388, update your request URL to `getDetailsJSON.php`, and try out the new version of the inventory page.



This looks just like the XML version... but it uses JSON as the data format. One more choice for Rob to look at...

Watch it!

Be sure you have JSON.php along with the JSON server-side script.

The server-side scripts for this chapter require JSON.php, which comes with this chapter's downloads. Be sure you have all those files before going on.

JSON.php is used by the server-side script in this chapter. It handles some PHP-specific issues that make dealing with JSON easier on the server.

Look, Joe's using the old specs for the site. Rob wants to send dynamic data, like group names and artists and hat sizes.



Frank, who thought he had the Les Paul guitar in the bag.

Jim just doesn't want to see Joe win.

Frank: You're right! His JSON code doesn't really help much in that case. His code depends on knowing the names of the object's properties.

Jim: Exactly. And there's no way it works with dynamic item descriptions, where the property names change.

Frank: Hmmm. You know, I was able to solve that problem with XML. I wonder if Joe—

Jim: No way, man. You've always read the name of properties from your element's tag names. His code has the property names as part of the code... they're not even parameters to search methods like `getElementsByTagName()`.

Frank: You're right. I'll bet this will hang him up pretty good.

I'm not done yet. There's got to be a way to handle dynamic properties...



BRAIN POWER

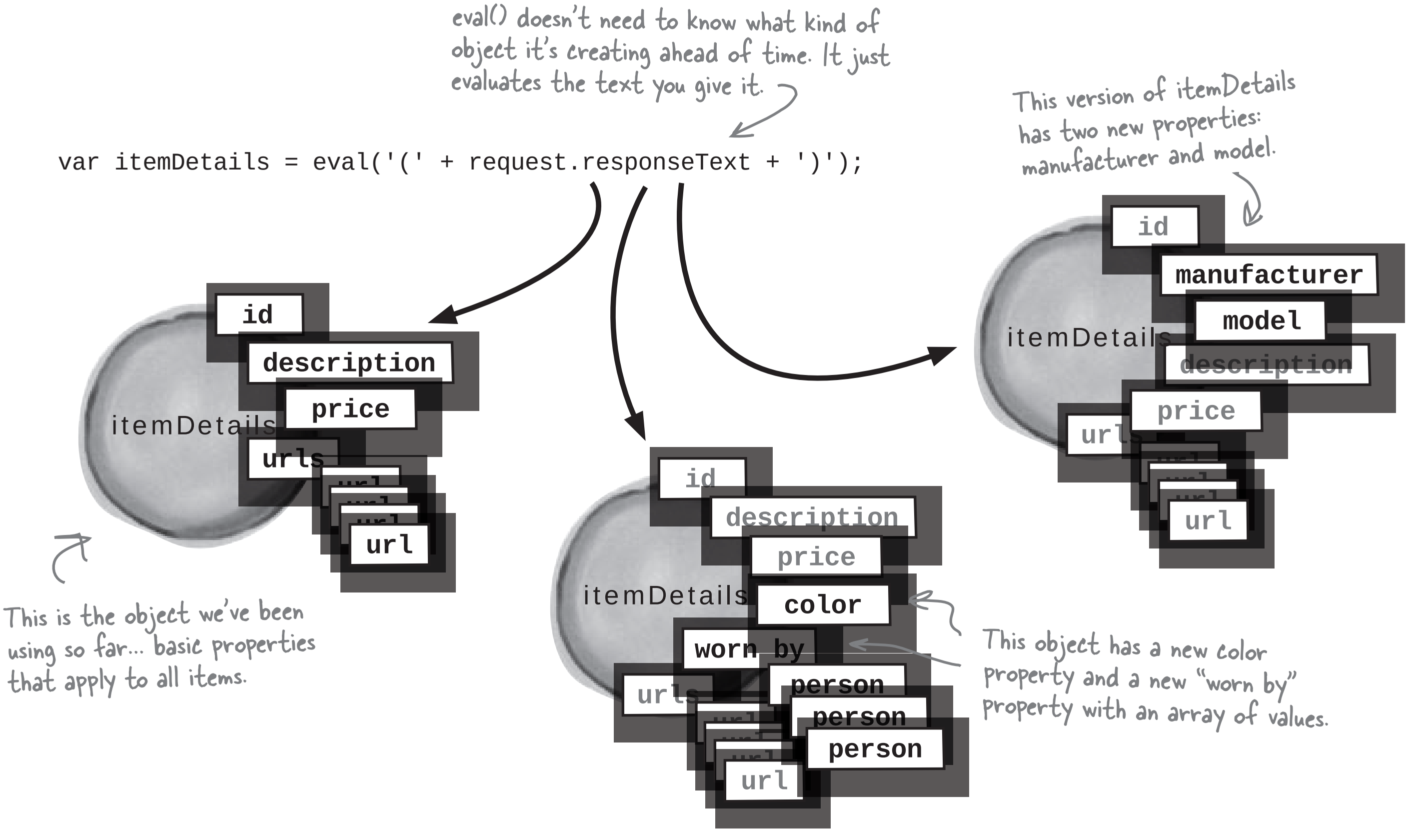
What would a JavaScript object that had to change based on the specifics of an item look like?

JavaScript objects are ALREADY dynamic... because they're not COMPILED objects

In compiled languages, you define your objects in a source file, like a `.java` or `.cpp` file. Then, you compile those files into bytecode. So once your program's running, you're stuck with the definitions that are compiled into bytecode. In other words, a `Car` object can't suddenly have a new `manufacturer` property without recompilation. That lets everyone who's using the `Car` object know what to expect.

.cpp is for C++ code.

JavaScript, however, *isn't* compiled; it's ***interpreted***. Things can change at any time. Not only that, but the objects the server sends are created at runtime, using `eval()`. So whatever's sent to our JavaScript, that's what we get in the `itemDetails` object.



We don't need to change our object at all! We just need to know how to figure out what's IN the object.

You can access an object's members... and then get an object's values with those members

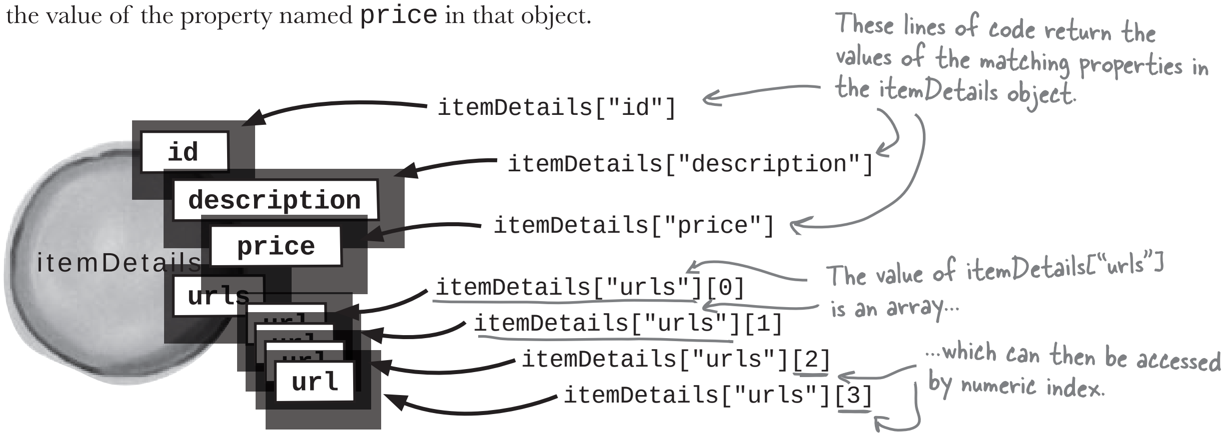
JavaScript will tell you what properties an object has. You just have to ask it, using the `for/in` syntax. Suppose you've got an object called `itemDetails`, and you want to know what properties `itemDetails` has. You'd use this code to get those properties:

```
for (var property in hero) {
  alert("Found a property named: " + property);
}
```

Pretty simple, right? So the variable `property` would have values like `id`, `description`, `price`, and `urls`.

But we don't want just the property names; we also want the *values* for each property. That's okay, though, because JavaScript lets you access an object's properties as if the object were an array. But instead of supplying an array index, like `itemDetails[0]`, you supply a property name, like `itemDetails["price"]`.

In other words, the value returned for `itemDetails["price"]` for an object is the value of the property named `price` in that object.



Exercise

You know what to do. Update your version of the inventory page to work with dynamic data from the server. You never know what you'll get... just that the server will return an object in JSON format with properties and values for those properties. Good luck!

Hold on there. I got the single-valued properties working, but how am I supposed to deal with arrays? How do I know if a property value, like urls, holds an array?



JavaScript does NOT give you a built-in way to see if a value is an array.

Dealing with dynamic data is tricky business. For instance, when you write in your code `itemDetails.urls`, you know that the value for that property will be an array. But what about `itemDetails[propertyName]`? Is the value for that property an array or a single value, like a string?

Unfortunately, JavaScript doesn't give you a simple way to check and see if a value is an array. You can use the `typeof` operator, but even for arrays, `typeof` returns "object," and not "array" like you might expect.

To help you out, here's a little Ready Bake Code that will tell you if a value is an array or not. Add this function to the end of `thumbnails.js`, and then see if you can finish up your exercise from the last page.

Ready Bake Code

isArray() returns true if you pass it an array value, and false if you pass it something else, like a string value.

```
function isArray(arg) {

  if (typeof arg == 'object') {

    var criteria = arg.constructor.toString().match(/array/i);

    return (criteria != null);

  }

  return false;

}
```

Arrays are all considered objects by JavaScript.

All arrays have a constructor with the word "array" in that constructor.

The "i" at the end means ignore case.

Any object that has that constructor is an array.

If the argument isn't an object, it's not an array.

Add this entire function to the end of thumbnails.js.



You know what to do. Update your version of the inventory page to work with dynamic data from the server. You never know what you'll get... just that the server will return an object in JSON format with properties and values for those properties. How did you do?

```
function displayDetails() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var detailDiv = document.getElementById("description");

      var itemDetails = eval('(' + request.responseText + ')');
      // Remove existing item details (if any)
      var children = detailDiv.childNodes;
      for (var i=children.length; i>0; i--) {
        detailDiv.removeChild(children[i-1]);
      }

      // Add new item details
      for (var property in itemDetails) {
        var propertyValue = itemDetails[property];
        if (!isArray(propertyValue)) {
          var p = document.createElement("p");
          p.appendChild(
               document.createTextNode(property + ": " + propertyValue));
          detailDiv.appendChild(p);
        } else {
          var p = document.createElement("p");
          p.appendChild(document.createTextNode(property + ":"));
          var list = document.createElement("ul");
          for (var i=0; i<propertyValue.length; i++) {
            var li = document.createElement("li");
            li.appendChild(document.createTextNode(propertyValue[i]));
            list.appendChild(li);
          }
          detailDiv.appendChild(p);
          detailDiv.appendChild(list);
        }
      }
    }
  }
}
```

Nothing's changed in this section... this just clears out existing content.

We can cycle through each property of the returned object.

Start by getting the property's value and seeing if that value is an array.

Remember to add isArray() to your code, or this JavaScript won't work.

Single-valued properties are easy. We just need a <p> and some text.

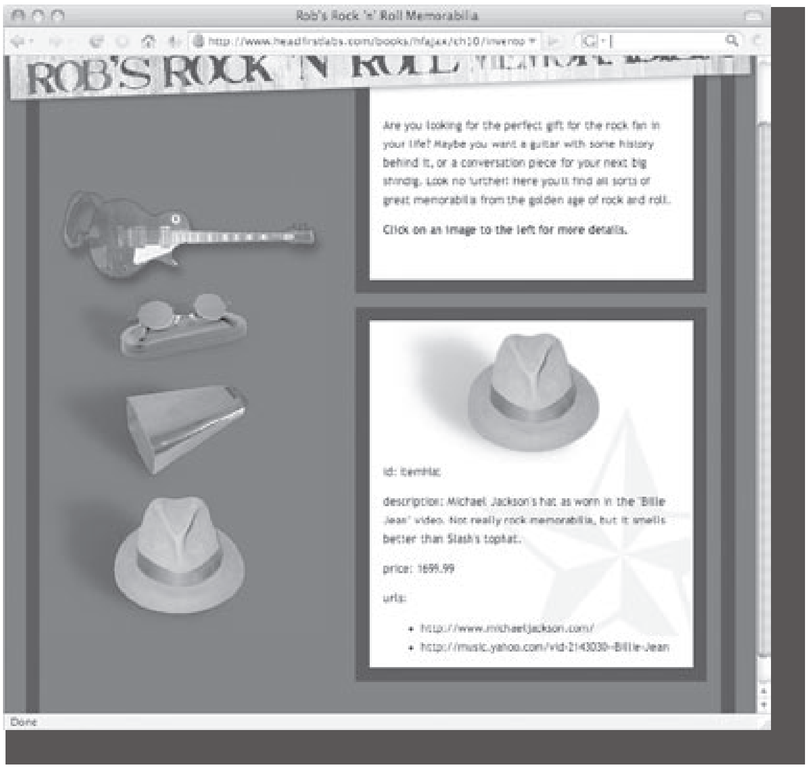
For multi-valued properties, we have to iterate through the array of property values.

For each value in the array, create a new <li> and add the value to it.

Test Drive

JSON testing, part deux

Now our code handles dynamic objects, values that might be strings *or* arrays, and should run like a dream. Let's see the new-and-improved JSON page...



Everything works... do you think Rob will be impressed?

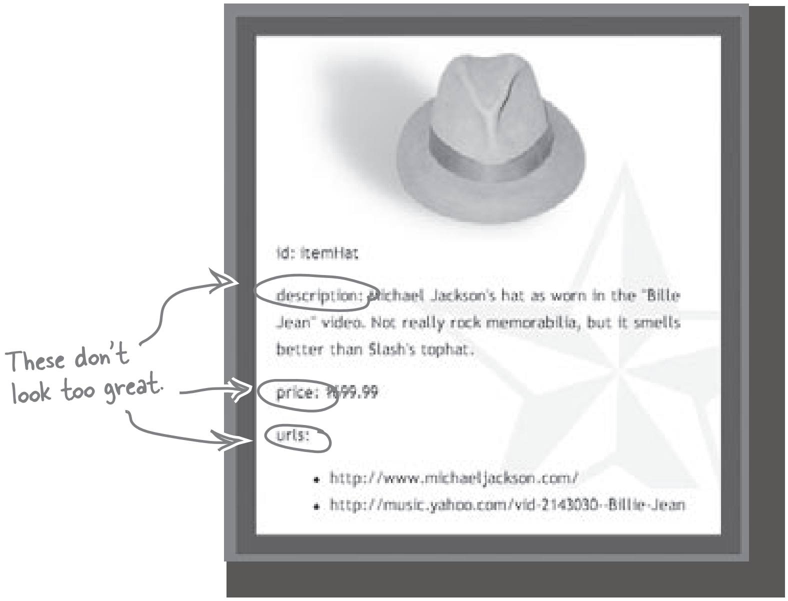
BRAIN POWER

Do you think isArray() belongs in thumbnails.js or utils.js? Put the function where you think it really belongs, and make any needed changes to the rest of your code now.



Good property names aren't usually good label names, too.

Take a close look at the property names for an item's description:



We've been printing out the property name and then the value for that property. But those property names look more like code than "human speak."

Not only that, but the ID of each item is showing, too. That could wind up being a real security bug down the line.

What would YOU do to fix these problems?

Those aren't the only problems. eval()s not safe to use like that... do you realize you're evaluating text from another source?



Joe: Well, yeah, that's kind of the point.

Frank: Do you really think that's a good idea? Just running code that someone else gave you?

Joe: I'm not running it, I'm evaluating it. Haven't you been paying attention?

Jim: Someone's getting cranky that their JSON solution isn't so easy...

Frank: Whether JSON's easy or not, you can't just evaluate that code blindly. What if it's malicious code, like a script that hacks the user's browser or something? Or it redirects their browser to a porn site?

Joe: Are you kidding me? It's Rob's server, for crying out loud!

Frank: What if it's not correct JSON? What if there's an error? Evaluating code with an error in it is going to generate errors for the users?

Jim: Sounds pretty dismal, Joe...

Joe: You're both just annoyed that I was gonna win that guitar.

Frank: Hey, safety first, man. I'm telling you, you can't go around using `eval()` on code that you don't have any control over.

Joe: Great. ***Now*** what am I gonna do?

eval() evaluates what you give it, WITHOUT regard for the results of that evaluation.

You ONLY have direct control of eval() code.

BRAIN BARBELL

Ajax request objects can only make requests to programs on the same server that the JavaScript creating the request was served from. Does that reduce, increase, or change the dangers of using eval() on a server's response text?

You need to PARSE the server's response, not just EVALUATE it

Calling `eval()` initiates a simple process: take a bit of text, and evaluate that text. What we need is an additional step. Suppose we could take a bit of text, and make sure it's actually JSON-formatted data. ***Then***, we could reasonably assume it's safe to evaluate that data, and turn it into a JavaScript object.

That extra step—parsing the data and making sure the data is JSON—protects us from two important potential problems:

1. We'll know that the data is safe to evaluate, and not a malicious script or program.

JavaScript code, or other scripts, won't pass a simple, "Is this JSON?" test.

2. We can be sure that not only is the data JSON, but it's *correctly formatted* JSON and won't cause our users any errors.

A parser can catch errors and report them, instead of just giving up and creating an error.

Fortunately for Joe (and us!), the JSON website at `http://www.json.org` provides a JSON parser that does all of these things, and more. You can download a script from json.org called `json2.js`, and then use this command to parse JSON-formatted data:

```
var itemDetails = JSON. parse( request.responseText );
```

You still need to assign the result of calling the parse() function to a variable.

This JSON object is created when json2.js is first loaded by the web browser.

parse() takes in a string and returns an object if the string is valid JSON-formatted data.

We can pass the server's response directly to JSON.parse().



Change your code to use JSON.parse().

The examples for Chapter 10 already include `json2.js` in the `scripts/` directory. Add a reference to this new script in `inventory.html`, and update your version of `thumbnails.js` to use `JSON.parse()` instead of `eval()`.

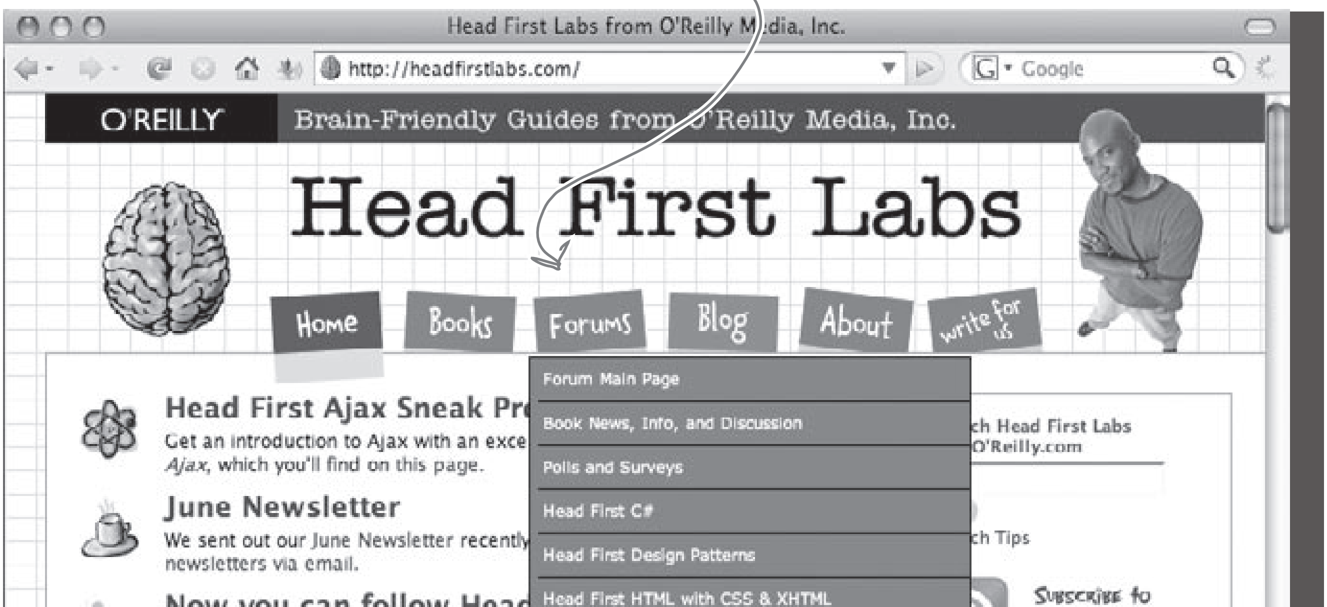
Put the reference to json2.js before the reference to thumbnails.js since the thumbnails script uses the json2 script.

There's still lots to do. Can YOU help Joe out?

- How could you avoid showing the ID of an item when a user clicks on that item?
- What about those labels? Can you figure out a way to show better, more- readable labels?
- What about those URLs? Can you figure out a way to format URLs as links (using <a> elements) so they're clickable?
- And besides all that, how do YOU think Rob's inventory page could be improved?
- Don't forget to use a JSON parser, instead of eval()!

Can you make Rob's inventory page even cooler using JSON? Build your best version of Rob's page, and submit your URL in the Head First Labs "Head First Ajax" forum. We'll be giving away cool prizes for the best entries in the coming months.

Visit us here, tell us about your entry, and give us a URL to check out what you've done (and how).



there are no Dumb Questions

Q: So I shouldn't ever use eval()?

A: `eval()` is an important part of JavaScript. If you need to pass textual data to another function for evaluation, or even between scripts, `eval()` is really helpful. However, `eval()` can be a problem when you're evaluating data that you can't control, like from someone else's program or server. In those cases, you won't know ahead of time exactly what you're evaluating. So in situations where you're not in control of all the data, stick to a parser or some other approach *other than* `eval()`.

Q: But a JSON parser keeps my code safe, right?

A: A JSON parser keeps your code safer than `eval()`, but that doesn't mean you can completely relax. When you're writing web code, security is *always* an issue. In the case of JSON data, `JSON.parse()` will ensure you've got valid JSON data, but you still don't know what that data actually *is*. So you may still need additional checks before using the data in the rest of your scripts.

Q: We didn't do any checks like that for Rob's page. Should we?

A: That's a good question. When you're reworking the app to help out Joe, think about the data you're getting. Could it be used maliciously? Do you think you need additional security checks?

Q: What about that json2.js script? Can I trust and rely on that code?

A: Now you're thinking like a web programmer! Anytime you use code from another source, like from `http://www.json.org`, you should thoroughly test out the code. We've done that testing here at Head First Labs, and `json2.js` is safe to use.

Q: Is it free, too? Do I have to pay anyone anything to use json2.js?

A: `json2.js` is free and open source. You can actually read through the source code at `http://www.json.org/json2.js`, and see what it does for yourself.

Q: So what about XML versus JSON? Which is better? And who won the guitar?

A: That's another good question. You've seen a lot of JSON and XML code now... which do you like best?

Security is ALWAYS a concern when you're programming for the web.

Always thorougly test any code that you don't have complete control over.

So which is the better data format?



Joe: I know about lots of things I don't like. Brussel sprouts, Melrose Place, velcro shoes... just because I know about something, doesn't mean I like it!

Frank: I just don't see what you really gain by using JSON. Maybe it's a little easier for *you* to use, but it's a pretty big pain when you get to dynamic data.

Jim: I don't know, Frank... I got really confused dealing with 2 DOMs at once.

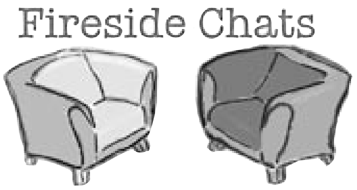
Frank: But XML is self-describing! We didn't have any of those property-names-as-labels issues with XML.

Joe: I still think JSON lets me think in JavaScript, not some other language.

Sharpen your pencil

What do YOU think? Below are two columns: one for XML, and one for JSON. Under each heading, write why you think that format is better. See if you can come up with ***at least*** 5 good arguments for XML, and 5 more for JSON.

XML	JSON



Tonight's talk: **XML and JSON go head-to-head on data formats and standardization**

XML: *(glares at JSON)*

JSON: Your time has finally come, XML. Tonight, the world is gonna see that you've lost a step, especially when it comes to JavaScript and asynchronous applications.

XML: I've heard that one before... but here I am, still the reigning data format in the world.

JSON: You're only at the top because people think that there's nothing else available. I know lots of people that can't stand you, XML... you're big and bloated, and a real pain to work with.

XML: I'm big because I can handle anything: product memorabilia, HTML, purchase orders... you throw it at me, I'll take care of it. No problem. You think a little pipsqueak can handle all those different types of data? I don't think so.

JSON: Maybe not, but I'm fast... a lot faster than you, most of the time.

XML: I'm plenty fast, especially if you use my attributes. And I'm versatile... I can do all sorts of things, like represent a math equation or a book.

JSON: Yeah, well, most of my users aren't too interested in sending math equations across the network. Besides, all those angle brackets? Ugh... anyone that knows arrays can start working with me without having to learn all that weird XML syntax.

XML: But can someone transform you into something else? Like with XSLT? Or what about web services... you're gonna tell me you can handle web services?

XML: **JSON:**

JSON: Wow, you've really been a bit overused, haven't you... you're missing the point, Bracket-Head. I don't care about all those things. I just care about getting information from a web page to a server and back, without a bunch of extra work... like having to crawl up and down a DOM tree. Know anyone who thinks *that's* fun?

XML: Uh, yeah. Hello? We've got a whole group of DOM experts out there these days, writing killer user interfaces. Did you see that Fifteen Puzzle? That was pretty cool, and it was only about 100 lines of code. Anyone that knows the DOM is ready to use XML, *today*!

JSON: Look, all developers really need is a lightweight data format that's easy to work with in JavaScript. And that's me, Big Boy, not you.

XML: What are all the servers going to think about this? You know, PHP and ASP.Net and Java... I don't see them lining up to throw their support to you and your "lightweight data format" spiel.

JSON: Well, I guess that's true... but there are libraries that those guys can use to work with me.

XML: Libraries? If they've got to use a library, why not use a standard like the Document Object Model?

JSON: I'm already standard in PHP 5. And who knows who's going to adopt me next?

XML: But here I am, being used right now, because I'm *already* a standard. At the end of the day, you're just one more proprietary data format. Maybe you've got a few more fans than comma-separated values, but I'll put an end to that.

JSON: Oh really? Let's see about that...

Sharpen your pencil Solution

What do YOU think? Below are two columns: one for XML, and one for JSON. Under each heading, write why you think that format is better. See if you can come up with ***at least*** 5 good arguments for XML, and 5 more for JSON.

XML	JSON

ANSWERS OBSCURED INTENTIONALLY

These are YOUR answers. It's up to you to decide between JSON and XML, based on the factors that are important to you.

Hop online and continue the XML vs. JSON discussion at the Head First Ajax forum on http://www.headfirstlabs.com.

11 forms and validation

Say what you meant to say

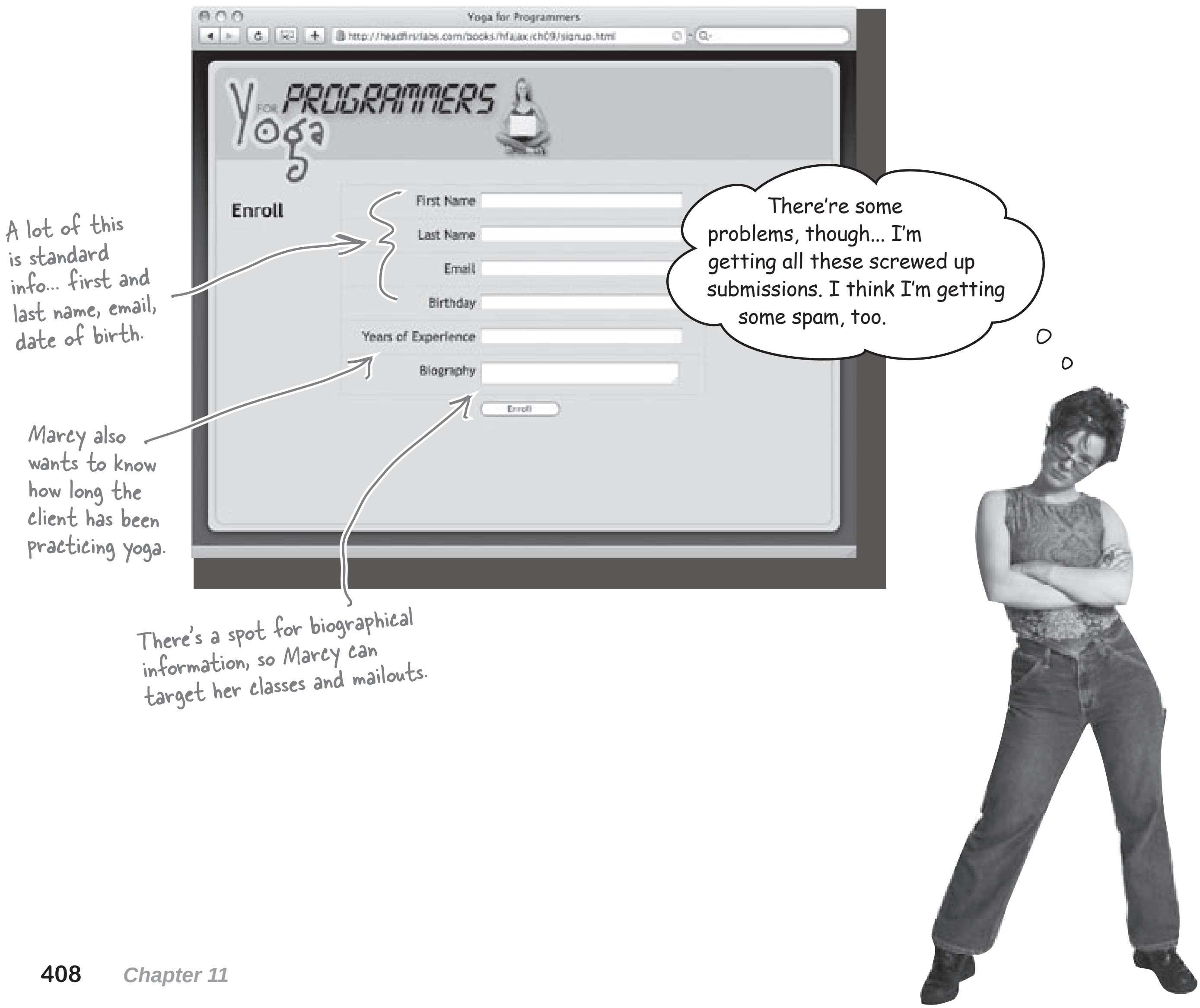


Everyone makes mistakes from time to time.

Give a human being a chance to talk (or type) for a few minutes, and they'll probably make at least one or two **mistakes**. So how do your web apps **respond to those mistakes**? You've got to **validate** your users' input and react when that input has problems. But who does what? What should your web page do? What should your JavaScript do? And what's the role of the server in **validation** and **data integrity**? Turn the page to answer all of these questions, and a lot more...

Marcy's Yoga for Programmers... a booming new enterprise

With her hip new site and super-fast response times, Marcy's Yoga for Programmers site has exploded. She's got some of Silicon Valley's highest-end clientele signing up daily. She's even added online enrollment, so once a potential client finds the perfect class, they can sign up right away:



Sharpen your pencil

Below are a few entries from Marcy's ever-growing customer database. There are some big problems... can you figure out what they are?

firstname	lastname	email	bday	yrs	bio
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
Bob	Brown		August 300	5	
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
F0b#2938					View my porn for free!!!! 192.72.90.234
Jones	Jane	www.myjob.com			
Gerry	MacGregor	mac@myjob	March 23, 1972	99	
Mary		mw@myjob.com			I've been doing yoga for 12 years
Bill	Bainfield	bb@myjob.com	5-27-69		

How many problems can you spot with this data?

1. ....
2. ....
3. ....
4. ....
5. ....
6. Gerry MacGregor isn't old enough to have been practicing yoga for 99 years.
7. ....
8. ....
9. ....
10. ....

Sharpen your pencil Solution

Below are a few entries from Marcy's ever-growing customer database. How many problems were you able to spot?

firstname	lastname	email	bday	yrs	bio
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
Bob	Brown		August 300	5	
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
F0b#2938					View my porn for free!!!! 192.72.90.234
Jones	Jane	www.myjob.com			
Gerry	MacGregor	mac@myjob	March 23, 1972	99	
Mary		mw@myjob.com			I've been doing yoga for 12 years
Bill	Bainfield	bb@myjob.com	5-27-69		

1. Susan Smith is registered twice.
2. Bob Brown didn't give his email address.
3. The F0b#2938 entry is spam, not a real client.
4. Jane Jones entered in a website URL, not an email address.
5. Gerry MacGregor's email isn't valid... he probably left off .com or .org.
6. Gerry MacGregor couldn't have been practicing yoga for 99 years.
7. Mary didn't enter in a last name.
8. Everyone's using a different format for their birthday.
9. There's information missing for Jane Jones, Bob Brown, and Bill Bainfield.
10. ....

Did you come up with any other problems?

Sharpen your pencil

Based on the data that Marcy's trying to gather, what sorts of things would you do to ensure she isn't having the sorts of problems you saw on the last couple of pages?

For each field below, write down what you think you need to check.

First name

Last name

E-Mail

Birthday

Years of Yoga

Biography

Sharpen your pencil Solution

Based on the data that Marcy's trying to gather, what sorts of things would you do to ensure she isn't having the sorts of problems you saw on the last couple of pages?

First name
This should be a required field.
Names should only have letters.

Do we allow initials? That might mean we can allow periods.

What about spaces? Those might be okay, too...

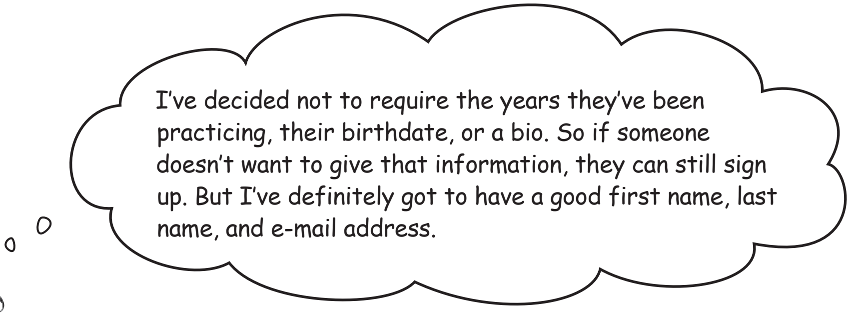
Last name
This should be a required field.
Names should only have letters.

E-Mail
This should be a required field.
We also need to make sure it's formatted like an e-mail.

Birthday
This should be a required field.
This should be some sort of consistent format, like MM-DD-YY, or something similar.

Years of Yoga
This should be a required field.
This should be a number, and be less than the years the person has been alive (calculated from their birthday).

Biography
This should be a required field.
Maybe we need a length limit?





The owner of the page knows their requirements better than you do.

No matter how good of a programmer you are, you're not an expert in your customer's business. So you probably won't always make good assumptions about things like required fields, the types of data that can be entered, or the format of that data.

For Marcy's site, Marcy is the customer, and you're the programmer.

It's best to come up with some basic ideas about validation, and then confirm and expand those ideas by talking to whomever actually owns the site or business that the site represents.

there are no Dumb Questions

Q: Is this gonna be another one of those, "Not really Ajax" chapters?

A: Yes and no. We'll be spending most of the chapter working on validation, not asynchronous requests. But figuring out how to actually get accurate requirements and validating data for those requirements applies to all software development, not just Ajax apps.



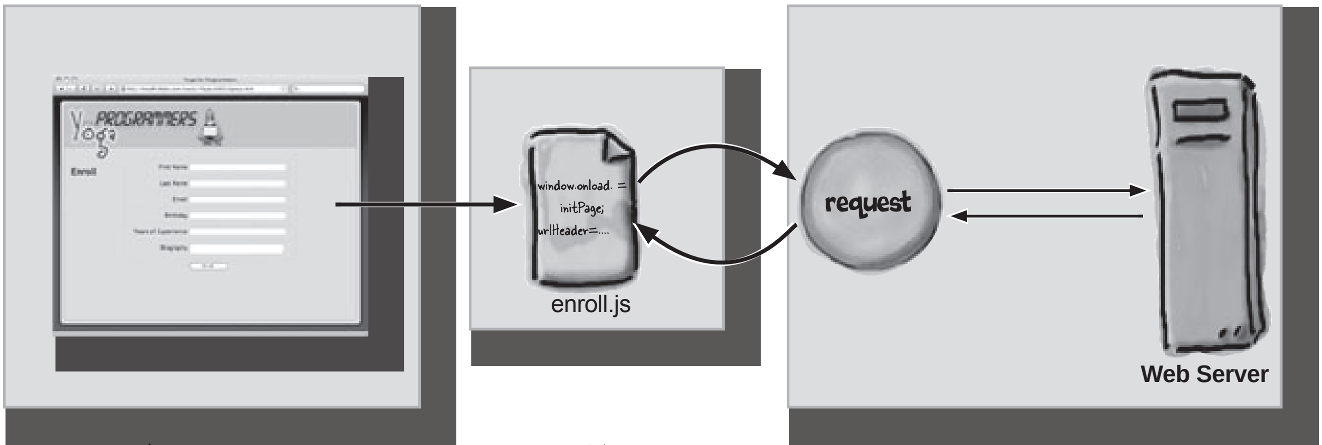
The customer defines the requirements, not the programmer.

The best way to build a site your customer loves is to build the site the your customer actually *wants*. Don't make assumptions about functionality... instead, ASK the customer how they want things to work.

Validation should work from the web page BACK to the server

Validation is usually a multi-step process. Some things you can catch by using certain controls on your web page, like a select box instead of a text field. You can catch other things with your client-side JavaScript, like the format of an email field. And still other things might need to go to the server to get validated, like seeing if a username's already taken.

The most effective way to handle multi-layered validation like this is to always validate as much as you can on the web page. Then, move to JavaScript, and validate as much as you can there. Finally, involve the server.



Web page

A web page can constrain data through specific controls, like length-limited text boxes and select boxes with only a few appropriate options.

Do as much as you can here to constrain data. Don't check for things in your JavaScript that you can restrict with form controls.

JavaScript

Your client-side JavaScript can check data formats, ensure data's entered into fields, and prevent submissions until a group of fields have data in them.

Try and never send the server data that's not formatted correctly. Let the server worry about business logic, not formatting.

Server

The server has access to the business data of your app. So it can check data consistency or perform other business logic that requires interaction with other data in the system.

The server should focus on the correctness of the data: based on existing data, is this new data correct and consistent?

Constraints → Validity → Consistency

Sharpen your pencil

Below is the XHTML for Marcy's current version of the enrollment form. What changes would you make, based on the things you wrote down on page 412, along with Marcy's comments?

Go ahead and mark your changes directly on the XHTML.

```
<html>
<head>
 <title>Yoga for Programmers</title>
 <link rel="stylesheet" type="text/css" href="css/yoga-enroll.css" />
</head>
<body>
 <div id="background">
  <h1 id="logo">Yoga for Programmers</h1>
  <div id="content">
   <h2>Enroll</h2>
   <form action="process-enrollment.php" method="post">
    <fieldset><label for="firstname">First Name</label>
     <input name="firstname" id="firstname" type="text" /></fieldset>
    <fieldset><label for="lastname">Last Name</label>
     <input name="lastname" id="lastname" type="text" /></fieldset>
    <fieldset><label for="email">Email</label>
     <input name="email" id="email" type="text" /></fieldset>
    <fieldset><label for="birthday">Birthday</label>
     <input name="birthday" id="birthday" type="text" /></fieldset>
    <fieldset><label for="years">Years of Experience</label>
     <input name="years" id="years" type="text" /></fieldset>
    <fieldset><label for="bio">Biography</label>
     <textarea name="bio" id="bio"></textarea></fieldset>
    <fieldset class="nolabel">
     <input type="submit" id="enroll" value="Enroll" />
    </fieldset>
   </form>
  </div>
 </div>
</body>
</html>
```

Sharpen your pencil Solution

Your job was to add constraints to the data Marcy collects from her customers by changing her XHTML. What did you come up with? Here's what we did.

```
<html>
<head>
 <title>Yoga for Programmers</title>
 <link rel="stylesheet" type="text/css" href="css/yoga-enroll.css" />
</head>
<body>
 <div id="background">
  <h1 id="logo">Yoga for Programmers</h1>
  <div id="content">
   <h2>Enroll</h2>
   <form action="process-enrollment.php" method="post">
    <fieldset><label for="firstname">First Name</label>
     <input name="firstname" id="firstname" type="text" /></fieldset>
    <fieldset><label for="lastname">Last Name</label>
     <input name="lastname" id="lastname" type="text" /></fieldset>
    <fieldset><label for="email">Email</label>
     <input name="email" id="email" type="text" /></fieldset>
    <fieldset><label for="birthday">Birthday</label>
     <input name="birthday" id="birthday" type="text" />
     <select name="month" id="month">
           <option value="">--</option>
           <option value="january">January</option>
           <option value="february">February</option>
           <!-- ... etc... -->
        </select>
       <select name="day" id="day">
          <option value="">--</option>
          <option value="1">1</option>
          <option value="2">2</option>
          <option value="3">3</option>
          <!-- ... etc... -->
```

There are a fixed number of values for birthday, so let's not use a text box, which allows bad entries.

Instead, we can use a select box for the month, and list the 12 possible month values...

...and another select for the day of the month, with all the possible day values.

Marcy told us she didn't want a birth year, so we didn't need to worry about that.

```
    </select>
    </fieldset>
    <fieldset><label for="years">Years of Experience</label>
     <input name="years" id="years" type="text" />
        <select name="years" id="years">
          <option value="">--</option>
          <option>none</option>
          <option>less than 1</option>
          <option>1-2</option>
          <option>3-5</option>
          <option>more than 5</option>
        </select>
    </fieldset>
    <fieldset><label for="bio">Biography</label>
     <textarea name="bio" id="bio"></textarea></fieldset>
    <fieldset class="nolabel">
     <input type="submit" id="enroll" value="Enroll" disabled="disabled" />
    </fieldset>
   </form>
  </div>
 </div>
</body>
</html>
```

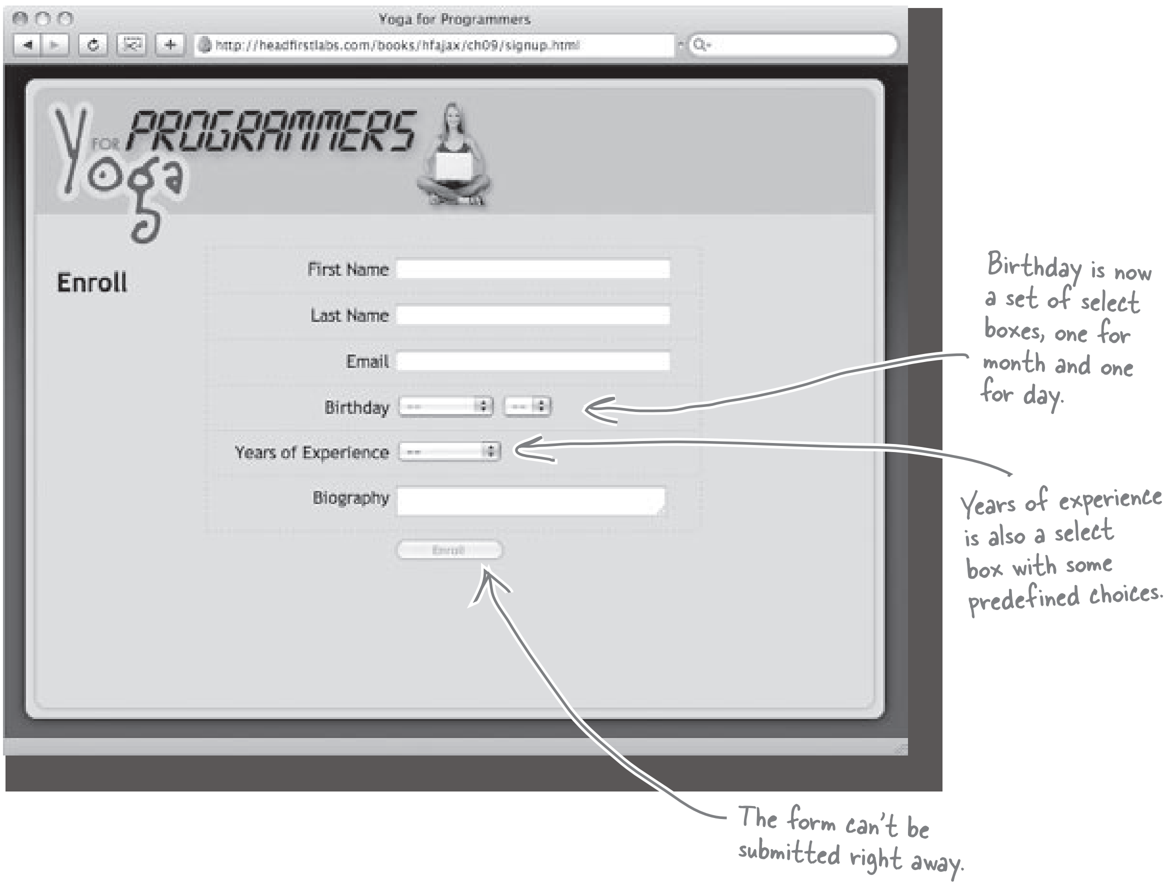
We can group the years of experience into useful ranges, and simplify things here, too.

We're going to need some JavaScript validation, so let's disable the Enroll button... we know people can't enroll without filling out some fields, so this protects the form from being submitted too soon.

Test Drive

See how many errors we can catch now...

Download or type in `signup.html`, and make the changes from page 416 and 417. Then load the page up in your browser. We've already knocked out a few of the problems Marcy was having:



there are no Dumb Questions

Q: Are you kidding? This isn't even JavaScript... it's just HTML. What gives?

A: It can definitely be a little boring to dig into XHTML if you'd rather be writing JavaScript and asynchronous requests. Then again, your coding gets a lot easier if you've got a good web page doing its job.

Q: So I should use select boxes whenever possible?

A: When it comes to data entry, that's a good principle. The more invalid or poorly formatted data that comes to your JavaScript, the more work your JavaScript has to do.

Q: What's the big deal with doing all of this in my JavaScript, and not messing with the XHTML web page?

A: Impatient customers are the big deal. It's often easy for you to code validation in your scripts, but customers don't like error messages. If you can make sure they enter data by using good controls, customers are less likely to need error messages from your validation code. That makes for a happier user experience, and that's always a good thing.

Q: Why did you disable the Enroll button in the HTML? Haven't we usually been doing that in an initPage() function, and calling initPage() from window.onload?

A: In earlier chapters, we've used `initPage()` to disable buttons, yes. You can certainly do the same thing here, or you can set the button to disabled in the XHTML. There's not a big difference in either approach, really.

One slight advantage to disabling the Enroll button in your XHTML, though, is that the XHTML now really does represent the initial state of the page. In other words, `initPage()` doesn't change the form as soon as it loads. That makes the XHTML a more accurate represention of the form at load-time. Still, it's not a big deal if you'd rather disable the button in an `initPage()` function.

Nobody enjoys an error message that says, "Hey, you screwed that up. Try again."

You can validate the FORMAT of data, and you can validate the CONTENT of data

We've been using the term validation pretty loosely. At the user's browser, we might make sure that the user enters their first name and birthday. That's one form of validation. At the server, we might make sure that the user's username isn't already taken. That's another form of validation.

In the first case, you're validating a data ***format***. You might make sure that a username is at least six characters long, or that there's a value for the first name field, or that an email address has an `@` sign and a `.com` or `.org` in it. When you're validating a data format, you're usually working with client-side code.



Validate the format of user data with JavaScript.

By using client-side code to validate data formats, you can let users know of problems quickly, without waiting on a server response.

Well-designed applications validate both the FORMAT and the CONTENT of user data.

Sometimes you've got to do more than just see how many characters a string is, or make sure an entry is really a month of the year. You may need to check data against your database to prevent duplicate entries, or run a computation that involves other programs on your network.

In those cases, you're validating the content of user data. And that's not something you can usually do at the client. You'll need to send the data to your server, and let programs on the server check out the data for validity.



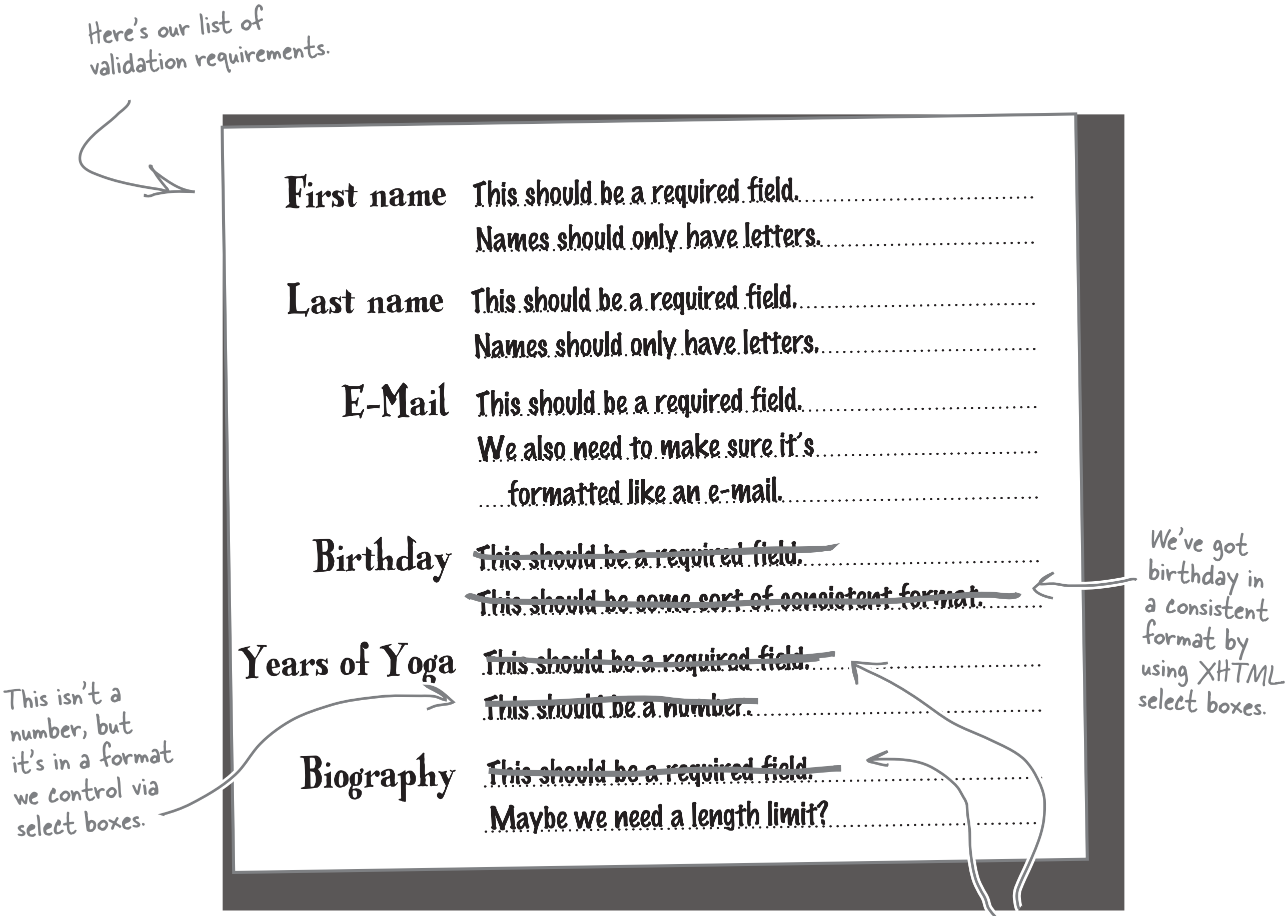
Validate the content of user data on the server.

You'll need your app's business logic to see if the content of user data is acceptable. Use server-side programs to let users know of problems with what they've entered.

You need BOTH types of validation to keep bad data out of your apps and databases.

We need to validate the FORMAT of the data from Marcy's enrollment page

Let's take another look at what we need to do to validate Marcy's page. For each field, we're actually just validating the format of the data. That means we should be able to do pretty much everything we need using JavaScript:





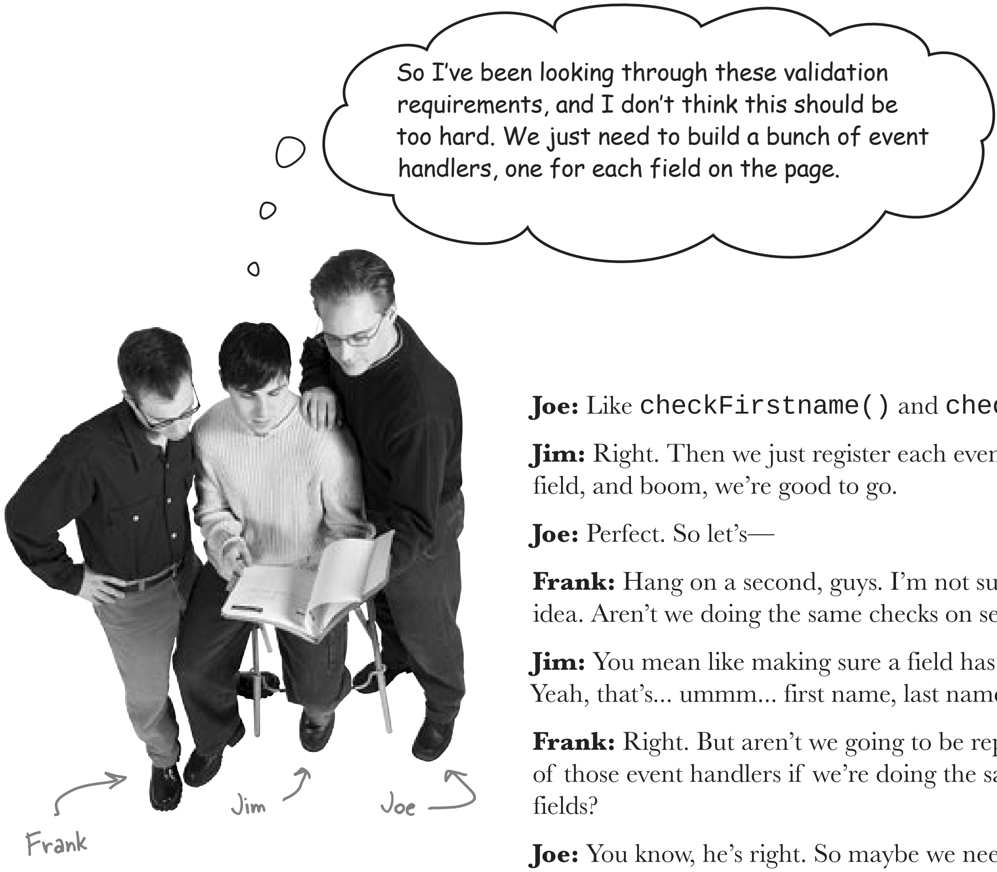
Exercise

We can use JavaScript to validate the format of all of these fields, but what *exactly* would you do next?

....

....

....



Joe: Like `checkFirstname()` and `checkLastname()`, right?.

Jim: Right. Then we just register each event handler to the right field, and boom, we're good to go.

Joe: Perfect. So let's—

Frank: Hang on a second, guys. I'm not sure that's such a good idea. Aren't we doing the same checks on several different fields?

Jim: You mean like making sure a field has a non-empty value? Yeah, that's... ummm... first name, last name, and email.

Frank: Right. But aren't we going to be repeating code in each one of those event handlers if we're doing the same checks for different fields?

Joe: You know, he's right. So maybe we need to have utility functions, like `fieldIsFilled()`, and we can call those from each event handler. So `checkFirstname()` and `checkLastname()` could just call `fieldIsFilled()` to see if those fields are empty.

Jim: Oh, that is better. So come one, let's get—

Frank: Wait a second. I still think we can do better. Why do we even need a `checkFirstname()` function?

Jim: Well, duh, that's got to call all the utility functions.

Joe: Hey, hang on, I think I see what Frank's getting at. What if we built the utilities to take in a field, and do their check?

Jim: But you'd still need something to call all the checks for each field. Like I said, `checkFirstname()`, or whatever...

Joe: But can't you assign multiple handlers to a single field?

Frank: That's it! So you could just assign each validation function to the field it applies to. Like this...

Don't Repeat Yourself: DRY

One of the core principles in software design is called DRY: don't repeat yourself. In other words, if you write a piece of code once, in one place, try to avoid writing that piece of code again in some other place.

When it comes to validation, that means we shouldn't write code that checks to see if a field is empty in two (or more!) places. Let's write one utility function, and then use that function over and over again:

```
function fieldIsFilled() {
  if (this.value == "") {
    // Display an error message
  } else {
    // No problems; we're good to go
  }
}
```

This function is generic. It can be applied as an event handler to any field.

Check to see if the field has no value...

...and then display an error or let the user continue.

Now you can assign this handler to several fields, for instance in an `initPage()` function:

```
document.getElementById("firstname").onblur = fieldIsFilled;
document.getElementById("lastname").onblur = fieldIsFilled;
document.getElementById("email").onblur = fieldIsFilled;
```

Because fieldIsFilled() isn't tied to a particular field, it can be used as a handler for multiple fields.



Don't Repeat Yourself

If the same code exists in two places, it's easy to change one bit of code and forget to change the other. By only writing code once, your app will be easier to maintain, as well as more modular. Don't repeat yourself!



Exercise

There's a pretty big problem with fieldIsFilled(). Can you figure out what it is, and fix it?

Hint: You might need another JavaScript file to correct the problems with fieldIsFilled().

This looks good, Jim, but I think there might be problems down the line, especially when you start assigning multiple handlers like this to the same object on a page.



Jim: What do you mean? I tried it out, everything works great.

Frank: But you're only assigning a single event handler to each field, right?

Jim: Right. And we've got our utility function, `addEventHandler()`, ready for when we need to add more handlers. So I'm all ready to handle multiple browsers and that whole `addEventListener/attachEvent` thing.

Frank: But you're using `this` in `fieldIsFilled()`...

Jim: Sure. What's the big... oh. Once we use `addEventHandler()`—

Frank: —`this` stops working. That's the problem.

Exercise Solution

Did you find the problem in fieldIsFilled()? If you assign multiple handlers to a field, you'll need to use addEventHandler()... and as soon as you use that function, the "this" keyword no longer works in your handlers. Here's how we fixed the problem.

```
function fieldIsFilled(e) {
  var me = getActivatedObject(e);
  if (me.value == "") {
    // Display an error message
  } else {
    // No problems; we're good to go
  }
}
```

We'll get an event object when our utility function, addEventHandler(), is used.

We need to get the activated object since "this" isn't reliable with multiple handlers registered to the same field.

This code will require utils.js, which is where we have getActivatedObject() and addEventHandler() coded.

there are no Dumb Questions

Q: Why do we need to use multiple event handlers again?

A: Because we're building a handler for each type of validation function, like checking to see if a field's value is empty or if a value is in an email format.

So for a single field, there might be several of those utility functions that should be assigned. For example, the firstname field shouldn't be empty, but it also should only contain alphabetic characters.

Q: So since we're using more than one event handler, we can't use this?

A: Indirectly, yes. Beceause we need multiple event handlers on some fields, we'll need to use the `addEventHandler()` utility method we wrote in `utils.js` earlier. And since we're using that approach to register handlers, we can't use `this` in those handlers.

Q: Wouldn't it be easier to use a shell function for each field, like checkFirstname(), and then call each individual validation function from that?

A: Not really. Switching from this to `getActivatedObject()` isn't a big deal (especially if you've got a set of helper functions, like we do in `utils.js`). Besides, we'd need even more functions. In addition to the validation functions, we'd need a wrapper for each field that just connected the field to all of its handlers.

Q: I don't think I got that DRY thing. Can you explain that again?

A: Sure. DRY stands for "Don't Repeat Yourself." DRY is a pretty well-known software design principle. DRY just means that you want a single piece of code appearing in one single place. So if you're checking a field for an empty value, you should have that code in one place, and other pieces of code that need that functionality then call that single bit of code.

If you follow DRY, you never have to change one piece of code in *more* than one place in your scripts. That means your code ends up being easier to change, maintain, and debug.

You can check out *Head First Object-Oriented Analysis and Design* for a lot more on DRY and other design principles.

Q: And how does DRY fit into Marcy's Yoga app?

A: Well, each of our validation functions is a single bit of code, in a single function. If we put that code into individual handlers, we might have duplicate code. So `checkFirstname()` might have code that checks for an empty field, but `checkLastname()` might have the same code. If you found a better way to do that bit of functionality, you'd have to make a change in two places—and that violates DRY.

Q: So you never repeat code, no matter what?

A: Every once in a while you'll have to violate DRY, but it's pretty rare. As a general rule, if you work really hard to follow DRY, you'll have better designed code. If you've tried but can't manage it, then don't worry too much. The point is to *try your best* to not repeat code, as that makes you design and write better code in the long run.

Code that doesn't repeat itself is easier to change, maintain, and debug.

Always try and write DRY code!

Let's build some more event handlers

`fieldIsFilled()` was pretty simple. Let's go ahead and write code for the other event handlers we'll need. We can build each just like `fieldIsFilled()`: using `getActivatedObject()`, we can figure out the activated object, and then validate the format of the field.

```
function fieldIsFilled(e) {
  var me = getActivatedObject(e);
  if (me.value == "") {
    // Display an error message
  } else {
    // No problems; we're good to go
  }
}

function emailIsProper(e) {
  var me = getActivatedObject(e);
  if (!/^[\w\.-_\+]+@[\w-]+(\.\w{2,4})+$/.test(me.value)) {
    // Display an error message
  } else {
    // No problems; we're good to go
  }
}
```

This handler checks an email format to make sure it's name@domain.com (or .org, .gov, etc.)

This is the regular expression for checking email formats from Head First JavaScript.

We'll work out what code we need for when there are errors and when there aren't any problems in just a little bit. For now, we can use these comments.

This handler checks a field to see if it only contains letters: from a-z, case-insensitive.

```
function fieldIsLetters(e) {
  var me = getActivatedObject(e);
  var nonAlphaChars = /[^a-zA-Z]/;
  if (nonAlphaChars.test(me.value)) {
    // Display an error message
  } else {
    // No problems; we're good to go
  }
}
```

Here's another regular expression. It represents all characters that are NOT between a to z, or A to Z. So all non-alphabetic characters.

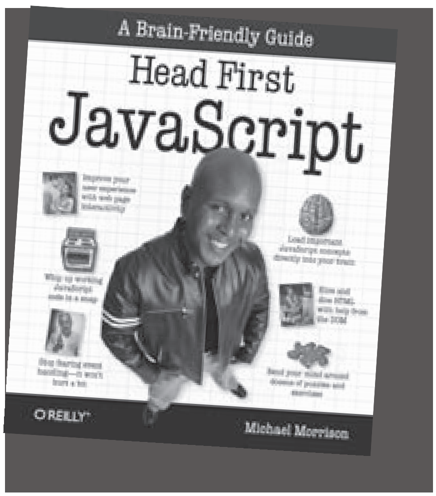
If any of these non-alphabetic characters are in the field's value, the value isn't all letters.

fieldIsNumbers() ensures a field has only numeric values in its value.

```
function fieldIsNumbers(e) {
  var me = getActivatedObject(e);
  var nonNumericChars = /[^0-9]/;
  if (nonNumericChars.test(me.value)) {
    // Display an error message
  } else {
    // No problems; we're good to go
  }
}
```

This expression grabs all characters NOT (using the ^ symbol) in the numbers 0 through 9.

For a lot more on regular expressions, check out Head First JavaScript.



Sharpen your pencil

Now that you've got event handlers, can you write initPage() for Marcy's app? Create a new script and save it as enroll.js. Add the event handlers above and your version of initPage(). Then reference enroll.js and utils.js in Marcy's XHTML.

Try and load the enrollment page. Does it validate your entries?

Sharpen your pencil Solution

Your job was to build an initPage() function for Marcy's yoga page, and use the handlers on the last two pages to validate user entry.

```
window.onload = initPage;

function initPage() {
  addEventHandler(document.getElementById("firstname"), "blur", fieldIsFilled);
  addEventHandler(document.getElementById("firstname"), "blur", fieldIsLetters);
  addEventHandler(document.getElementById("lastname"), "blur", fieldIsFilled);
  addEventHandler(document.getElementById("lastname"), "blur", fieldIsLetters);
  addEventHandler(document.getElementById("email"), "blur", fieldIsFilled);
  addEventHandler(document.getElementById("email"), "blur", emailIsProper);
}
```

We get each field...

...we want to validate when users move out of the field...

...and finally, we assign the handler for the field.

Several fields have multiple handlers assigned.

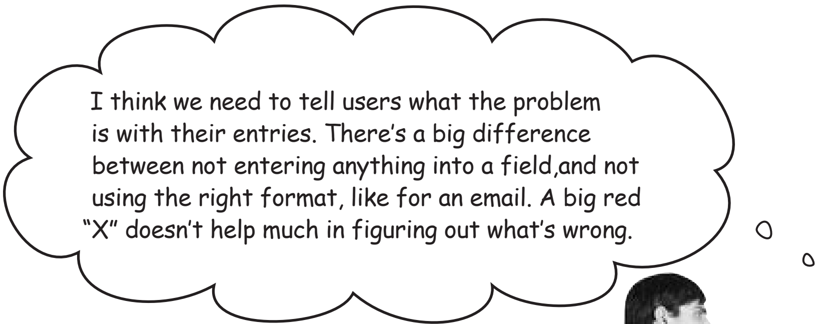
You said to test these, but how? Right now, we just have comments for when there's a problem... should we put in alert() statements to let the user know when there's a problem?



An alert() stops EVERYTHING... and users don't like to stop.

Using an `alert()` is pretty heavy-handed. That little popup brings everything on the page to a crashing halt. Earlier, we used some icons to let the user know what's going on. But there was a problem with that approach, especially if you try and apply it to what we're doing with Marcy's page.

Why doesn't a simple approved or denied icon work for Marcy's page? What would you do differently?





Jim: That makes sense. So for each error, we can just display a message related to that error. Like, "Please enter a first name" or "E-mails should be in the format name@domain.com".

Frank: Yeah. That seems a lot more user-friendly.

Joe: But that won't be so easy—.

Frank: Right, you see it, don't you?

Jim: What?

Joe: Well, we moved to generic handler functions. Those functions don't know about which field they're testing, so they won't know what error message to display.

Frank: Yeah. We need some way to have a set of error messages associated with each field. And then figure out a way to look up the right error message.

Joe: What about the activated object? We've got that in our handlers, so what if we use the object to look up an error message?

Jim: Hey, I've got an idea. Can we just have some sort of name/value thing, where there's a name of a field, and the value for that field is an error message?

Frank: I like that... I think that would work. So we lookup the error based on the name of the field, which we've got from the activated object.

Joe: But aren't there multiple problems that can occur for each field? We need more than one error message per field.

Frank: Hmmm. So we need a key for each field, and then a set of errors and corresponding messages for that. Right?

Jim: How the heck do we do that in JavaScript?

RETURN of SON of JavaScript

In the last chapter, server-side programs used JSON to represent complex object structures. But JSON isn't just for the server-side! Anytime you need to represent name-to-value mappings, JSON is a great solution:

This is the variable name for this object.

The value for itemDetails.id is "itemShades."

```
itemDetails = {
  "id" : "itemShades",
  "description" : "Yoko Ono's sunglasses. ...",
  "price" : 258.99,
  "urls" : ["http://www.beatles.com/",
            "http://www.johnlennon.com/",
            "http://www.yoko-ono.com/"]
}
```

The value for itemDetails.urls is an array of values.

The value of a property can be another JavaScript object

You've already seen properties have string values, integer values, and array values. But a property can also have *another* object as its value, again represented in JSON:

```
itemDetails = {
  "id" : "itemShades",
  "description" : "Yoko Ono's sunglasses. ...",
  "price" : 258.99,
  "urls" : {
    "band-url": "http://www.beatles.com/",
    "singer-url": "http://www.johnlennon.com/",
    "owner-url": "http://www.yoko-ono.com/"
  }
}
```

Curly braces signal another object value.

This time, the value of the urls property is another JSON-represented object.

itemDetails.urls.band-url

itemDetails.urls.singer-url

itemDetails.urls.owner-url

You can just "tack on" another dot operator to get to these nested values.



JSON Magnets

Can you use all the magnets below to build a set of mappings? You should have each field represented, and for each field, a set of mappings from a specific type of error to a message for that error.

```
var _______________ = {
  "________________" : {
    "______________": "____________________________________________________________",
    "______________": "____________________________________________________________"
  },
  "______________" : {
    "______________": "____________________________________________________________",
    "______________": "____________________________________________________________"
  },
  "______________" : {
    "______________": "____________________________________________________________",
    "______________": "____________________________________________________________"
  }
}
```

required

firstname

Only letters are allowed in a first name.

lastname

letters

Please enter in your e-mail address.

Please enter in your first name.

letters

email

Please enter your e-mail in the form 'name@domain.com'.

required

Only letters are allowed in a last name.

required

Please enter in your last name.

format

warnings



JSON Magnet Solutions

Can you use all the magnets below to build a set of mappings? You should have each field represented, and for each field, a set of mappings from a specific type of error to a message for that error.

warnings is the variable name for the overall object.

There's a specific error message for each field, and for each warning for that field.

```
var warnings = {
  "firstname" : {
    "required": "Please enter in your first name.",
    "letters": "Only letters are allowed in a first name."
  },
  "lastname" : {
    "required": "Please enter in your last name.",
    "letters": "Only letters are allowed in a last name."
  },
  "email" : {
    "required": "Please enter in your e-mail address.",
    "format": "Please enter your e-mail in the form 'name@domain.com'."
  }
}
```

There's a top-level mapping for each field that we have validation on.

There's a second-level mapping for each type of error that can happen on each field.

Let's warn Marcy's customers when there's a problem with their entry

With a warnings object full of useful messages, we can add warnings to Marcy's page. Here's what we've got in each event handler validation function:

1. The field, via an activated object, that we need to validate.
2. A specific type of problem that occurred (for example, we know whether a field was empty or invalidly formatted).

Based on that information, here's what we need to do in our warning:

1. Figure out the parent node of the field that there's a problem with.
2. Create a new `<p>` and add it as a child of that field's parent node.
3. Look up the right warning, and add that warning as text to the new `<p>`, which will cause the browser to display the warning on the form..

Here's a `warn()` function that handles this for Marcy's form:

In our handler functions, we can pass in the field and type of problem.

This evaluates the string that matches the field and warning type.

```
function warn(field, warningType) {
  var parentNode = field.parentNode;
  var warning = eval('warnings.' + field.id + '.' + warningType);
  if (parentNode.getElementsByTagName("p").length == 0) {
    var p = document.createElement("p");
    field.parentNode.appendChild(p);
    var warningNode = document.createTextNode(warning);
    p.appendChild(warningNode);
  } else {
    var p = parentNode.getElementsByTagName("p")[0];
    p.childNodes[0].nodeValue = warning;
  }
  document.getElementById("submit").disabled = true;
}
```

This check is to see if there's already a <p> that we can add the warning to.

If not, create a new <p> and text node with the right warning.

This "else" is for when there's already a <p> to add the warning to.

If there's a problem, make sure the "Enroll" button can't be clicked.

Exercise

We've done a lot over the last few pages, and before you test everything out, there are several steps you need to make sure you've taken. Here's what you need to do:

- [ ] Add the warnings variable from page 432 into your enroll.js script. You can put the variable anywhere outside of your functions, at the same "level" as your window.onload event handler assignment.
- [ ] Add the warn() function from page 433 into enroll.js, as well.
- [ ] Update each of your validation functions, like fieldIsFilled() and fieldIsLetters(), to call warn() when there's a problem. You should pass the warn() function the activated object, and a string, like "required" or "format." You can figure out which strings to use for the warning type by looking at the values in the warnings variable on page 432.

there are no Dumb Questions

Q: How does warn() know what field it's adding a warning message to?

A: Each validation function knows what field it's validating, because of `getActivatedObject()`. So when the handler function calls `warn()`, that function passes the activated object on to `warn()`.

Q: And what about the warning type? Where does that come from?

A: The warning type is specific to the event handler function. `fieldIsFilled()` would have a warning type of "required," because that's what that function is essentially checking for: to see if a required field has a value.

Each handler should pass on a warning type that matches one of the pre-defined values from the warnings variable, like "required" or "letters" or "format."

Q: What's all that parentNode stuff?

A: We want to add the warning just *under* the actual input box. If we get the parent of the input box (the field), then we can add another child of that same node with the warning. The result is that the warning message becomes a sibling of the input field itself... and displays right under the field.

Q: And the warning message is from the warnings variable?

A: Exactly. We put that message in a `<p>`, as a child of the field's parent node.

Q: What's going on with that eval() line? That's a little confusing to me...

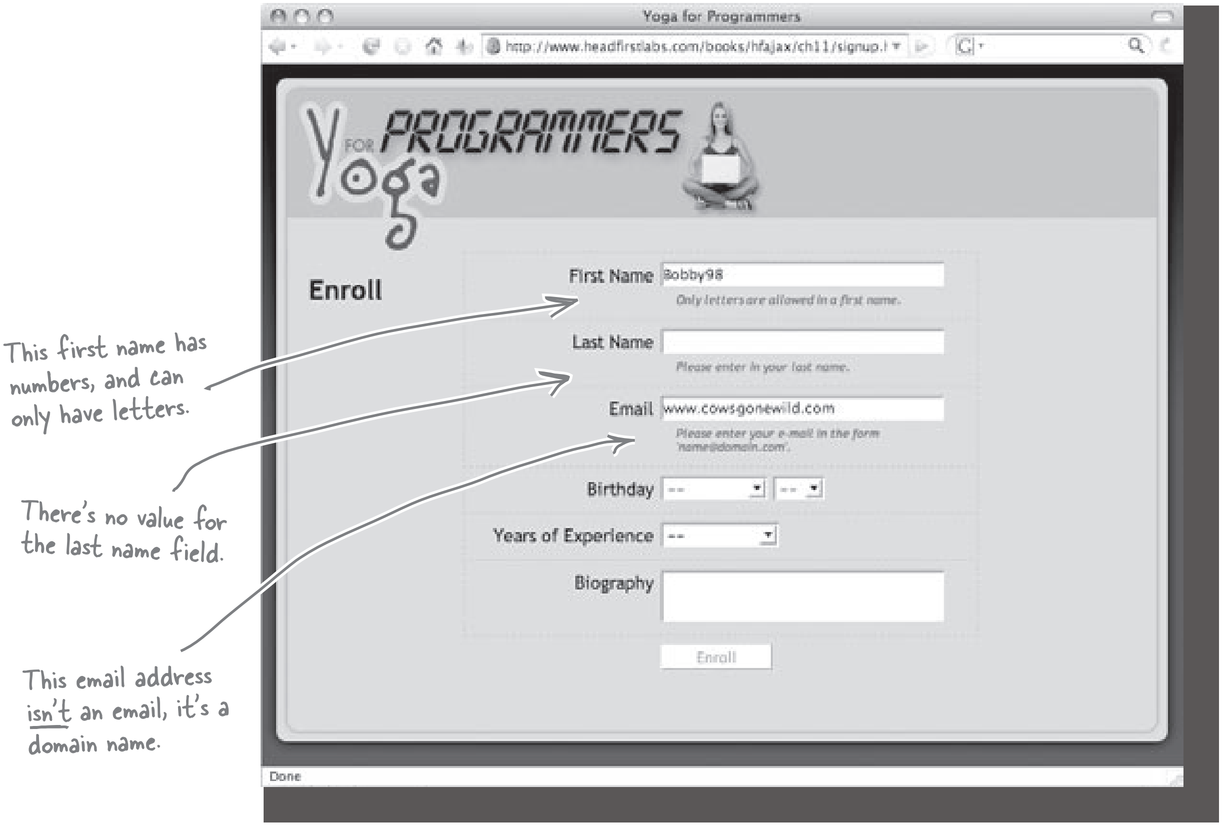
A: First, look at what's being evaluated: `'warnings.' + field + '.' + warningType`. That might come out to 'warnings.firstname.required' or warnings.email.format'. Each of those maps to an error message, which is what we want.

So to evaluate the expression `'warnings.firstname.required',` we run `eval()` on that expression. The result is the matching error message, which we can then show on the enrollment form.

Test Drive

It's time for more error counting.

Make sure you've done everything on the checklist on page 434, and then reload Marcy's enrollment page. Try out several "bad" combinations of data: skip entering a value for a few fields, enter in a bad email address, try numbers in the name fields. What happpens?



BRAIN POWER

How can we figure out when all fields are valid, and it's safe to allow users to click "Enroll"?

Wait a second... we spent all that time last chapter talking about how dangerous eval() is. And now we're using it again? Haven't you been paying attention?



eval() is safe to use when you CONTROL the data you're evaluating.

In Chapter 10, we were evaluating data from a server-side program. We didn't write that program, and weren't even able to look at the source code. That makes that code unsafe to evaluate. We just weren't sure that the code would be valid JSON, and would be safe to run on a user's browser.

But with the `warnings` variable, we *created* the code we're evaluating. So there's no danger. In fact, we can test things out, and if there is a problem, we just make a change to `warnings`. So it's perfectly safe to run `eval()` on code you're in control of.

```
eval('warnings.' + field.id + '.' + warningType);
```

```
eval('warnings.firstname.letters');
```

```
"Only letters are allowed in a first name."
```



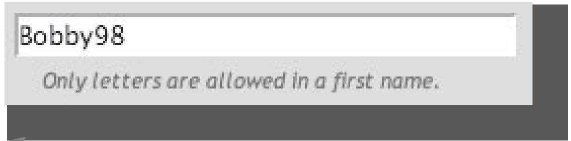
There's nothing unsafe about this eval() line because we control the variable it evaluates.

If you don't warn(), you have to unwarn()

There's one big problem with Marcy's enrollment page: how do we get rid of those error messages when there's ***not*** a problem? Here's what our error handlers look like right now:

```
function fieldIsFilled(e) {
  var me = getActivatedObject(e);
  if (me.value == "") {
    // Display an error message
    warn(me, "required");
  } else {
    // No problems; we're good to go
  }
}
```

The warn() function takes care of displaying errors on the form.



If there's not a problem, we need to remove any error messages.

IF there's a warning, get rid of it

Let's build an `unwarn()` function. The first part is pretty simple: for the field that's passed in, we just need to see if there's a warning. If so, we can get rid of the warning. If there's not a warning, we don't need to do anything:

```
function unwarn(field, warningType) {
  if (field.parentNode.getElementsByTagName("p").length > 0) {
    var p = field.parentNode.getElementsByTagName("p")[0];
    var currentWarning = p.childNodes[0].nodeValue;
    var warning = eval('warnings.' + field.id + '.' + warningType);
    if (currentWarning == warning) {
      field.parentNode.removeChild(p);
    }
  }
}
```

We only need to remove a warning if there's at least one <p> with a warning already in place.

Figure out which warning type we're unwarning for.

We only remove a warning if it matches the warningType passed in to unwarn().

If the warning types match, remove the warning.

Exercise

unwarn() isn't complete yet. The function still needs to figure out if the Enroll button should be enabled or disabled. Can you write code that figures out if there are any warnings being displayed? If so, Enroll should be disabled; otherwise, users can click Enroll to submit the form.

Hint: if you need a refresher, the XHTML for the enrollment page is on page 416.

Exercise Solution

unwarn() isn't complete yet. The function still need to figure out if the Enroll button should be enabled or disabled. Your job was to write code that figures out if there are any warnings being displayed? If so, Enroll should be disabled; otherwise, users can click Enroll to submit the form.

```
function unwarn(field, warningType) {
  if (field.parentNode.getElementsByTagName("p").length > 0) {
    var p = field.parentNode.getElementsByTagName("p")[0];
    var currentWarning = p.childNodes[0].nodeValue;
    var warning = eval('warnings.' + field.id + '.' + warningType);
    if (currentWarning == warning) {
      field.parentNode.removeChild(p);
    }
  }
  var fieldsets =
    document.getElementById("content").getElementsByTagName("fieldset");
  for (var i=0; i<fieldsets.length; i++) {
    var fieldWarnings = fieldsets[i].getElementsByTagName("p").length;
    if (fieldWarnings > 0) {
      document.getElementById("enroll").disabled = true;
      return;
    }
  }
  document.getElementById("enroll").disabled = false;
}
```

All the <p> warnings are children of <fieldset> elements, so let's get all those <fieldset>'s.

For each <fieldset>, we can see if there are any <p> child elements.

This is equivalent to seeing if there are any warnings since each warning is in a <p>.

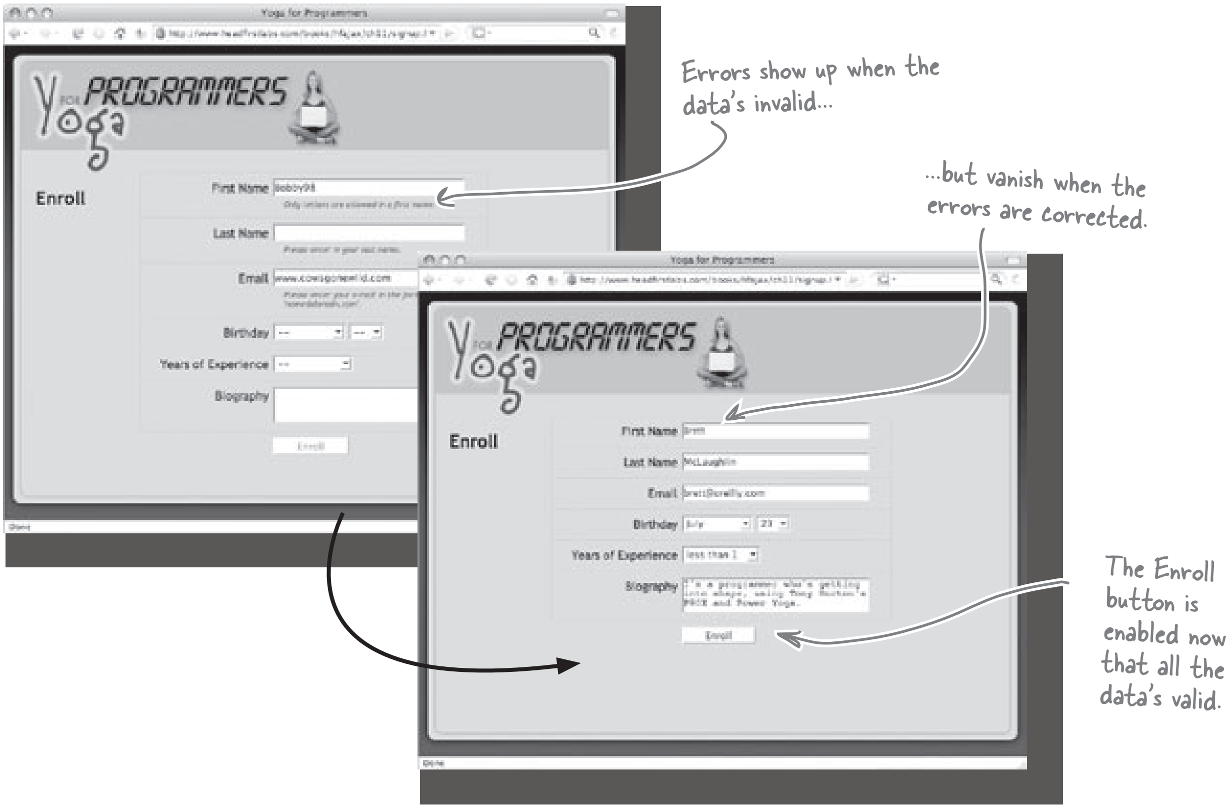
If there are any warnings, disable Enroll and return.

If there aren't any warnings, the form is okay... enable Enroll.

Test Drive

Turn warnings on AND off.

Time to take the enrollment form for another test drive. In each of your validation handlers, add a line that calls `unwarn(me);` if there's not a validation problem. Looking pretty good, right?





Validation is hard, thankless work... and <u>EVERY</u> application needs validation.

Getting data right on a form is often boring, and takes a long time to get right. But, validation is incredibly important to most customers. Take Marcy: without good data, she can't enroll people in classes, she can't send out mailings, and she can't get new business.

Multiply that by all the web apps that you're getting paid to develop, and validation becomes critical. And while Marcy's enrollment form isn't making asynchronous requests, it's still a web application that's typical of the things you'll have to work on as a web developer. Not many programmers can make a living ***only*** working on asynchronous requests.

So take the time to get validation on your pages right. Your customers will love you and their businesses will flourish... and that means more work, better paychecks, and less middle-of-the-night, "It's broken!" calls.

Every application needs validation!

Sharpen your pencil

Below is Marcy's database and the problems we found way back on page 410. Next to each problem, make a note about whether your changes to the enrollment form have fixed that problem yet.

firstname	lastname	email	bday	yrs	bio
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
Bob	Brown		August 300	5	
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
F0b#2938					View my porn for free!!!! 192.72.90.234
Jones	Jane	www.myjob.com			
Gerry	MacGregor	mac@myjob	March 23, 1972	99	
Mary		mw@myjob. com			I've been doing yoga for 12 years
Bill	Bainfield	bb@myjob.com	5-27-69		

1. Susan Smith is registered twice. ....
2. Bob Brown didn't give his email address. ....
3. The F0b#2938 entry is spam, not a real client. ....
4. Jane Jones entered in a website URL, not an email address. ....
5. Gerry MacGregor's email isn't valid... he probably left off .com or .org. ....
6. Gerry MacGregor couldn't have been practicing yoga for 99 years. ....
7. Mary didn't enter in a last name. ....
8. Everyone's using a different format for their birthday. ....
9. There's information missing for Jane Jones, Bob Brown, and Bill Bainfield. ....
10. ....

Sharpen your pencil Solution

We've added a lot of validation... but what problems have we really solved? Your job was to figure out what problems our validation is preventing.

firstname	lastname	email	bday	yrs	bio
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
Bob	Brown		August 300	5	
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
F0b#2938					View my porn for free!!!! 192.72.90.234
Jones	Jane	www.myjob.com			
Gerry	MacGregor	mac@myjob	March 23, 1972	99	
Mary		mw@myjob.com			I've been doing yoga for 12 years
Bill	Bainfield	bb@myjob.com	5-27-69		

We don't have anything to handle this yet.

1. Susan Smith is registered twice.
2. Bob Brown didn't give his email address.
3. The F0b#2938 entry is spam, not a real client.
4. Jane Jones entered in a website URL, not an email address.
5. Gerry MacGregor's email isn't valid... he probably left off .com or .org.
6. Gerry MacGregor couldn't have been practicing yoga for 99 years.
7. Mary didn't enter in a last name.
8. Everyone's using a different format for their birthday.
9. There's information missing for Jane Jones, Bob Brown, and Bill Bainfield.
10.

Required fields are handled now.

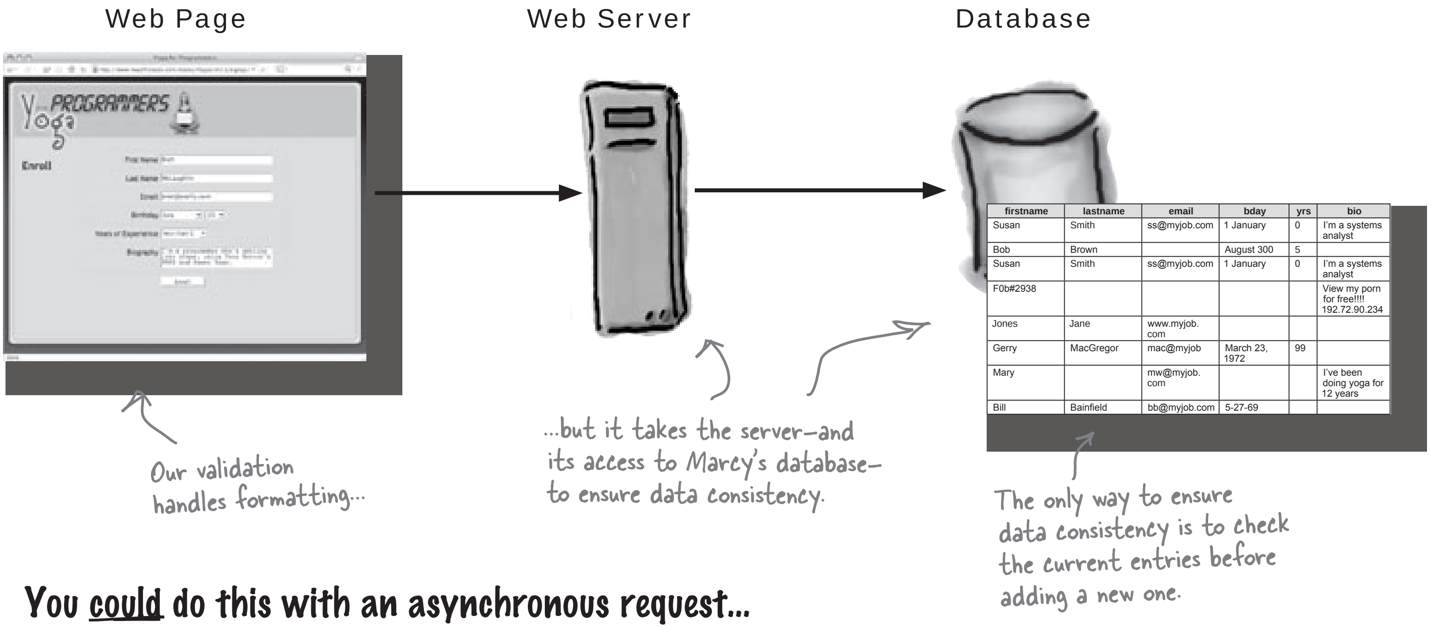
Our formatting requirements for names and emails take care of this.

The XHTML changes to the page keep these from occurring.

Between Marcy's updated requirements and our validation, this is no longer a problem.

Duplicate data is a SERVER problem

The only problem we've got left is when someone enters in their information twice, like Susan Smith on the last page. But it's going to take a server-side program to handle that sort of problem... the server would need to take an entry, and compare it with existing entries in Marcy's customer database.



firstname	lastname	email	bday	yrs	bio
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
Bob	Brown		August 300	5	
Susan	Smith	ss@myjob.com	1 January	0	I'm a systems analyst
F0b#2938					View my porn for free!!!! 192.72.90.234
Jones	Jane	www.myjob.com			
Gerry	MacGregor	mac@myjob	March 23, 1972	99	
Mary		mw@myjob.com			I've been doing yoga for 12 years
Bill	Bainfield	bb@myjob.com	5-27-69		

You could do this with an asynchronous request...

Suppose we build a server-side program to take a user's information, and check Marcy's customer database to see if that user already existed. We could request that program using an asynchronous request in our JavaScript. Then, once the server returned a response, we could let the user know that their data's been accepted.

...but what's the benefit?

The only problem is that there's nothinig for the user to do while they're waiting. We're probably using at least their first name, last name, and email to check against the database, so at most, a user could keep entering in their birthdate and bio. But even those aren't required fields...

It's really better to let the server check the user's information when the user tries to enroll, and issue an error then. Since duplicate users aren't a huge problem right now, you're better off saving a ton of extra code, and simply letting the server handle reporting a problem to the user.

Sometimes, it's best to let the server handle problems synchronously.

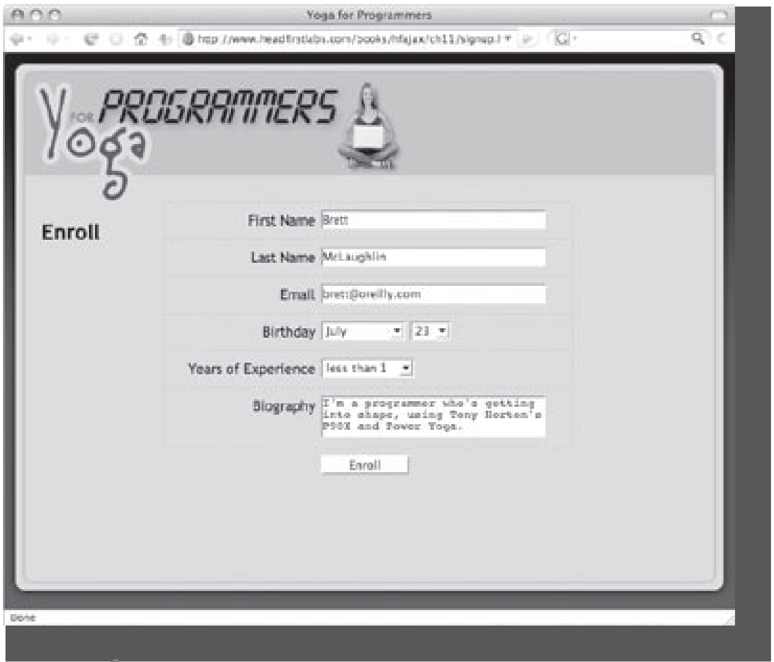
Not every web app needs asynchronous requests and responses!

So we're done now, right?

That's right. We've handled all of Marcy's validation problems, and she's going to have her server-side guys take a look at preventing duplicate data. In fact, let's see how Marcy likes her new enrollment page...

Wow, this is fantastic. I've been getting twice as many signups every week since going online, and I'm not getting any weird data problems. Care to sign up yourself?





Another web programming success story to add to your portfolio.

12 post requests

Paranoia: It's your friend



Someone's watching you. Right now. Seriously.

Freedom of Information Act? Isn't that called the Internet? These days, anything a user types into a form or clicks on a web page is subject to **inspection**. Whether it's a network admin, a software company trying to learn about your trends, or a malicious hacker or spammer, your ***information isn't safe unless you make it safe***. When it comes to web pages, you've got to **protect your users' data** when they click Submit.

There's a villain in the movies

Just when we thought that we'd solved all of the web world's problems, it looks like one of our earlier customers is back... and he's not happy.

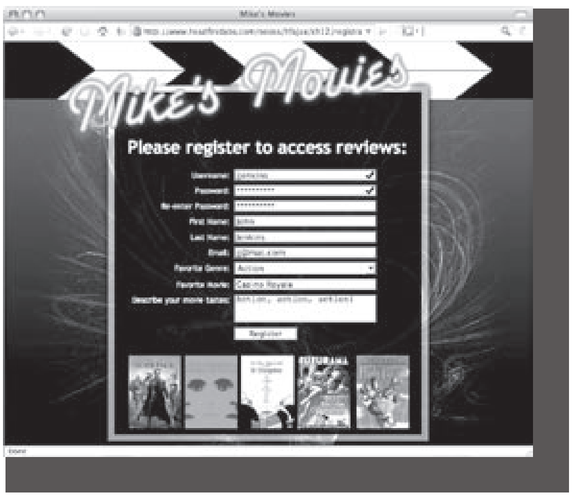
I've got issues. I'm starting to get customer complaints... they're all getting spammed, and they think it's because they registered for my movie review site. This is your code... you better fix it!



Your code... your problem!

Mike, of Mike's Movies fame, has another problem. It doesn't seem like his customers getting spammed is really related to the registration form we built for Mike, but since we built that form, he's blaming us. Welcome to web development.

So what do you think is going on? Is it possible that spammers are getting Mike's customer email addresses because of something we did—or didn't do—on the enrollment form?



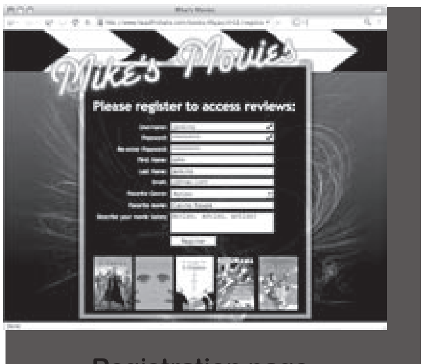
Remember Mike? We built his movie review registration page.

Sharpen your pencil

What's going on with Mike's registration page? Do we have anything to do with his customers getting spammed?

Below is Mike's page and server. Your job is to draw all the interactions between them. Be sure to include what's passing between the web page and the server.

Don't worry about the specifics of any particular user. Just write what fields and data is being sent back and forth.



Registration page



Web server

Do you think we have anything to do with the problems that Mike's customers are complaining about?

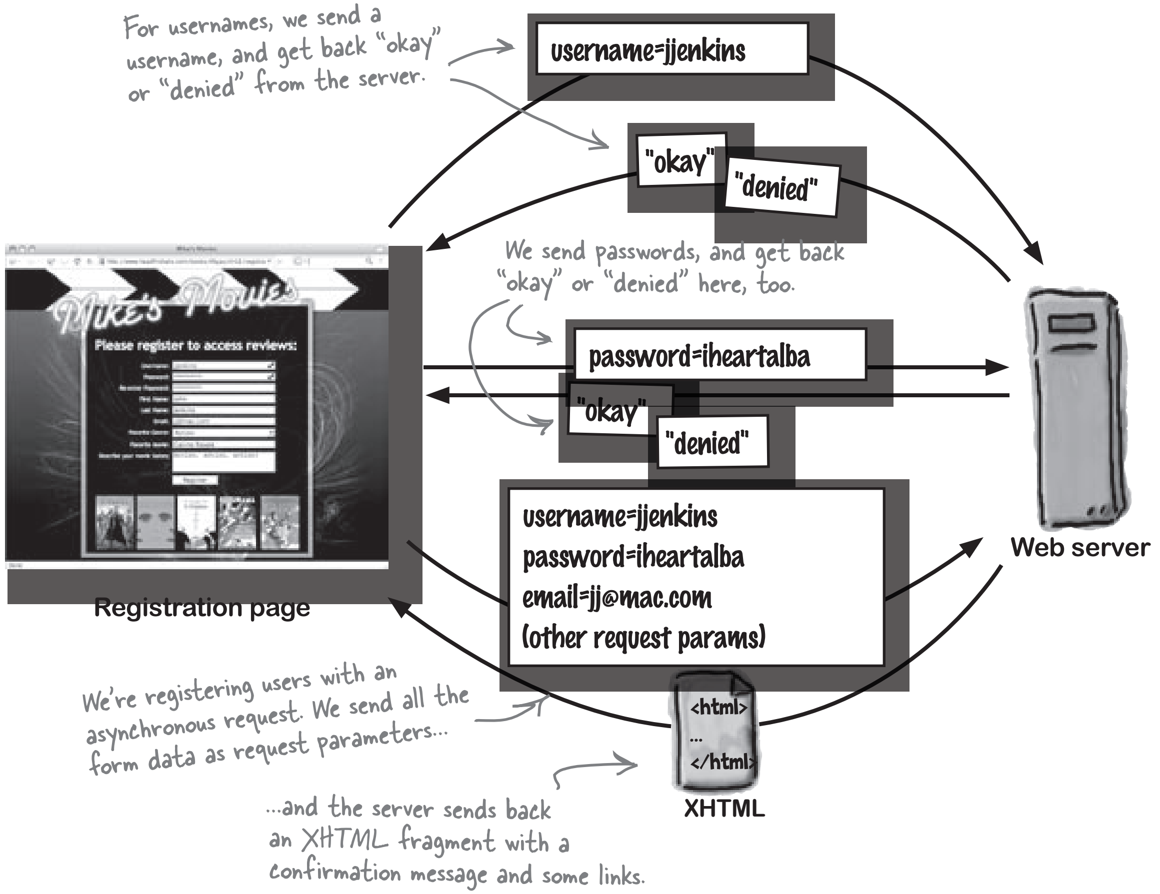
....

....

....

Sharpen your pencil Solution

Your job was to draw all the interactions between Mike's page and server. Be sure to include what's passing between the web page and the server. Here's what we came up with.



Do you think we have anything to do with the problems that Mike's customers are complaining about?

Maybe! We're sending the user's email across the network... is that safe? Can someone get that email and spam that user?

Hmmm... maybe Mike's problem IS something that's our code's fault.

GET requests send request parameters across the network as clear text

We're using a GET request to send all of a user's information to the server:

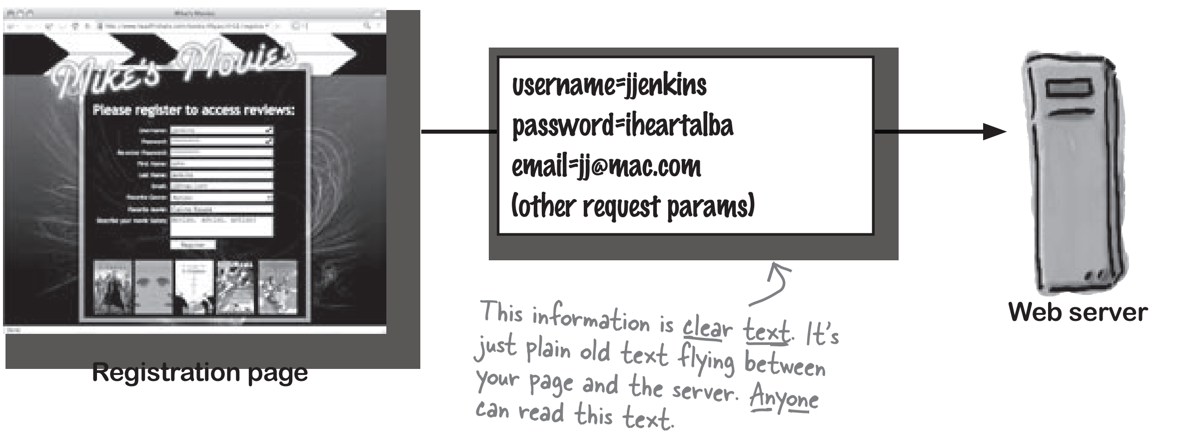
registerUser() sends a user's information using an asynchronous request.

```
function registerUser() {
  t = setInterval("scrollImages()", 50);
  document.getElementById("register").value = "Processing...";
  registerRequest = createRequest();
  if (registerRequest == null) {
    alert("Unable to create request.");
  } else {
    var url = "register.php?username=" +
      escape(document.getElementById("username").value) + "&password=" +
      other request parameters...;
    registerRequest.onreadystatechange = registrationProcessed;
    registerRequest.open("GET", url, true);
    registerRequest.send(null);
  }
}
```

Here's where we tell the request object to use a GET method for sending the request.

Clear text is text... in the clear!

When parameters are sent using a GET request, those parameters are just text moving across the network. And that text is sent ***in the clear***. In other words, anyone listening to your network can pick up that text.



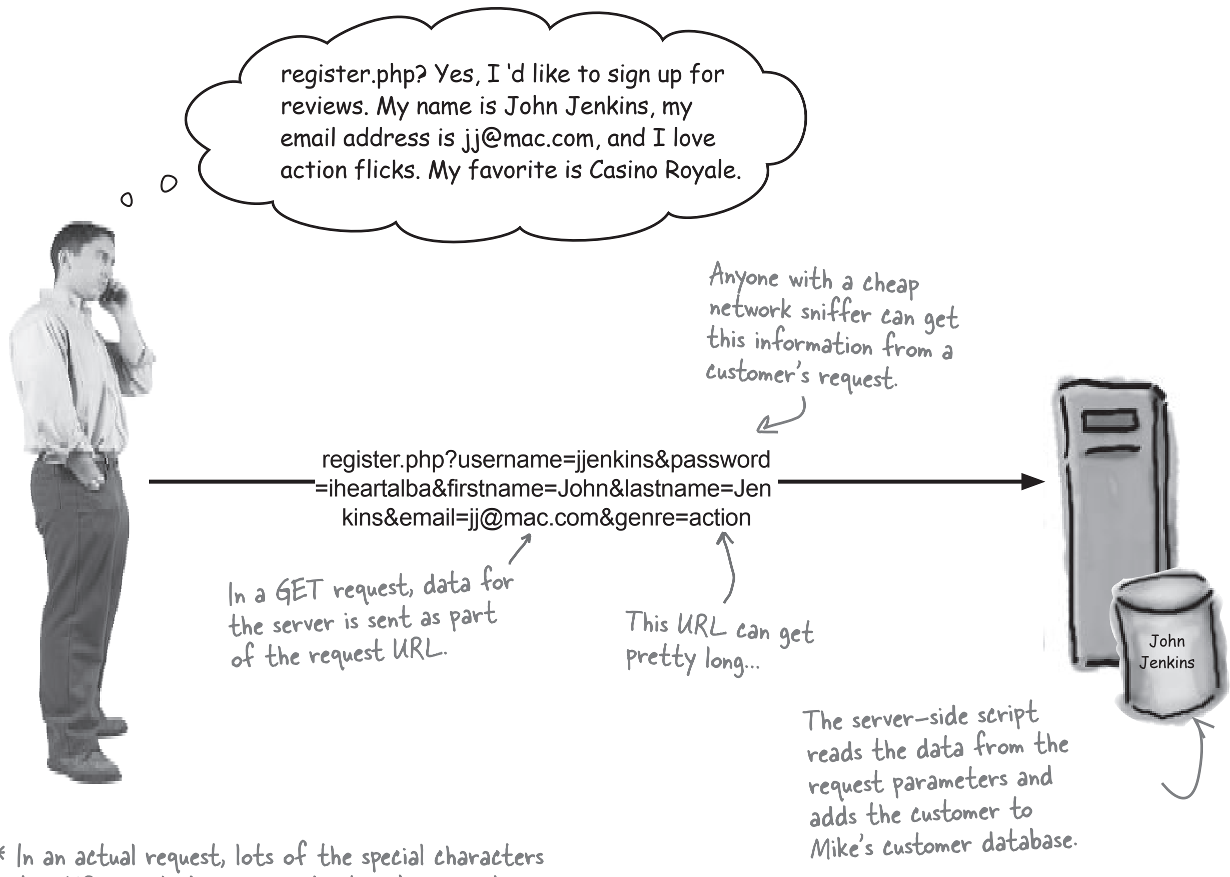
POST requests DON'T send clear text

What we need is a way to send that same data, but to avoid the data going across the network as clear text. That way, people can't snoop around and find Mike's customers' email addresses. That should take care of his spam problem once and for all.

Fortunately, that's just what POST requests do. They send their request data in a ***different way*** than GET does. Let's take a look:

GET requests send data in the request URL

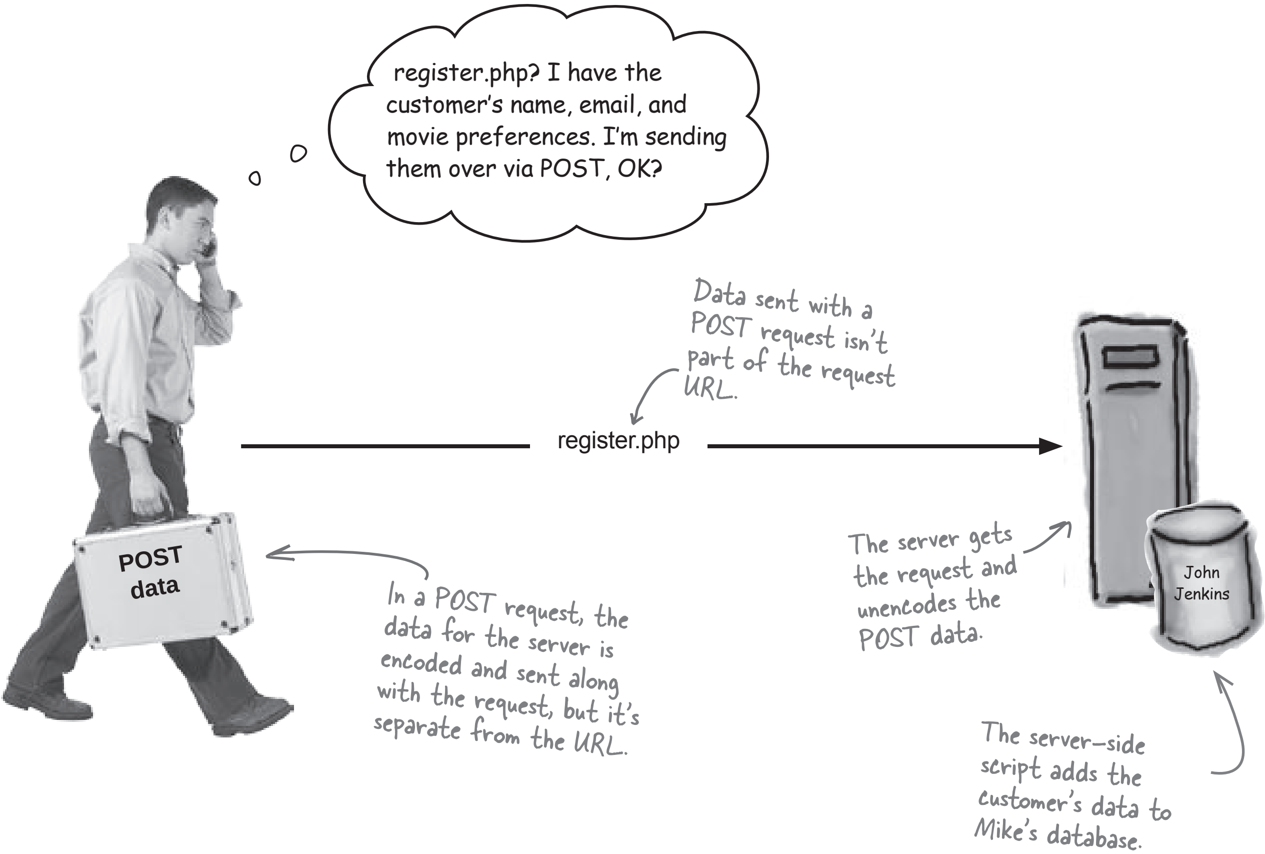
GET requests send data to the server as part of the request URL, using request parameters that are part of the actual URL.



* In an actual request, lots of the special characters in this URL would be encoded by the JavaScript escape() function. We've left it unencoded, though, to make it a little easier to understand.

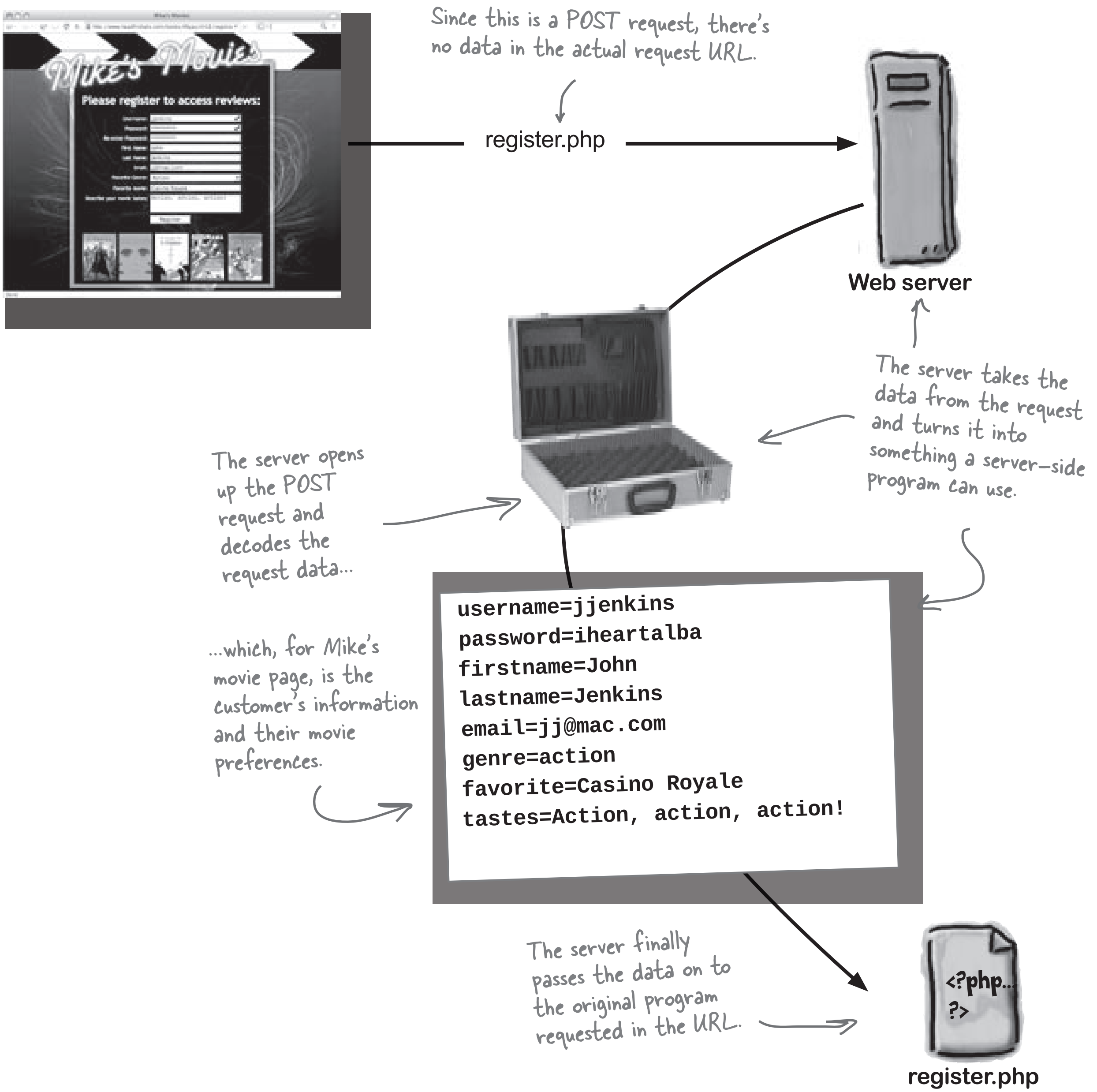
POST requests send data separate from the request URL

In a POST request, data that has to be sent to the server is kept separate from the URL. So there's no lengthy URL with data in it, and no clear text customer data is sent over the network.



The data in a POST request is ENCODED until it reaches the server

Once a web server gets a POST request, it figures out what type of data it has received, and then passes that information on to the program in the request URL.



there are no Dumb Questions

Q: So POST requests are more secure than GET requests?

A: Yes. There's one additional step that goes into packaging up POST data: the data is encoded in the browser and decoded on the server. Still, decrypting POST data isn't foolproof. Determined hackers can unencode your POST data, although it takes a lot more work than grabbing request parameters from the URL of a GET request.

If you really want to secure your request, you'll have to use a secure network connection, like SSL. But that's a little beyond what we're covering in this book.

Q: So if POST is still insecure, how will that help Mike's customers?

A: Most spammers are looking for the easiest targets possible. Most of the time, a single bit of trouble—like unencoding POST data—is all it takes to send spammers and hackers looking for an easier target. With Mike's site, moving to POST takes a little bit of effort, but will probably protect his site from the majority of malicious attacks.

A little bit of security on the Internet goes a long way.

Encoding your request data will cause most hackers to look for an easier target somewhere OTHER than on your web site.

Q: So are you saying that POST is safe and GET is unsafe?

A: Not really. "Safe" and "unsafe" are pretty relative terms, and it's impossible to predict all the ways something can go wrong. But sending data to the server using POST takes an extra step to protect that data. Sometimes that one step is the difference between your users getting your monthly newsletter, and those same users getting a spammer's porn mail.

Q: So why not send every request using POST?

A: There's really no need to. For one thing, encoding and unencoding data takes a bit of processing time. Besides that, GET is fine for sending shorter, non-private data. Also, if you use POST for everything, your users won't benefit from tools like Google Accelerator, and some search engine spiders might not pick up your links.

Q: And to send a POST request, all we have to do is put the request data in the send() method instead of the URL?

A: Exactly. You send the data in exactly the same format. You can pass name/value pairs to the request object's `send()` method, almost exactly like you did when you were sending a GET request.

Q: That's it? There's nothing else to do?

A: Well, let's try it out on Mike's page and see what happens...

send() your request data in a POST request

In a GET request, all the request data is sent as part of the request URL. So you build long URLs, like `register.php?username=jjenkins&password=...` But since request data isn't sent as part of the URL for a POST request, you can put all the data directly into the `send()` method of yor request object:

```
function registerUser() {
  t = setInterval("scrollImages()", 50);
  document.getElementById("register").value = "Processing...";
  registerRequest = createRequest();
  if (registerRequest == null) {
    alert("Unable to create request.");
  } else {
    var url = "register.php";
    var requestData = "username=" +
      escape(document.getElementById("username").value) + "&password=" +
      escape(document.getElementById("password1").value) + "&firstname=" +
      escape(document.getElementById("firstname").value) + "&lastname=" +
      escape(document.getElementById("lastname").value) + "&email=" +
      escape(document.getElementById("email").value) + "&genre=" +
      escape(document.getElementById("genre").value) + "&favorite=" +
      escape(document.getElementById("favorite").value) + "&tastes=" +
      escape(document.getElementById("tastes").value);
    registerRequest.onreadystatechange = registrationProcessed;
    registerRequest.open("POST", url, true);
    registerRequest.send(requestData);
  }
}
```

The request URL is just the name of the program on the server. No request parameters.

You don't need to precede yor request data with a question mark (?) in a POST request.

Instead of adding this data to the request URL, let's store it in a string variable.

Use the same ampersand character (&) to separate parameters.

This is a POST request now.

The request data is sent as a string and passed to the send() method of the request object.

there are no Dumb Questions

Q: Why don't I need a question mark?

A: The question mark (?) separated a server-side program name, like register.php, from the request data name/value pairs. Since you're not appending the request data to the program name, you don't need that question mark anymore.

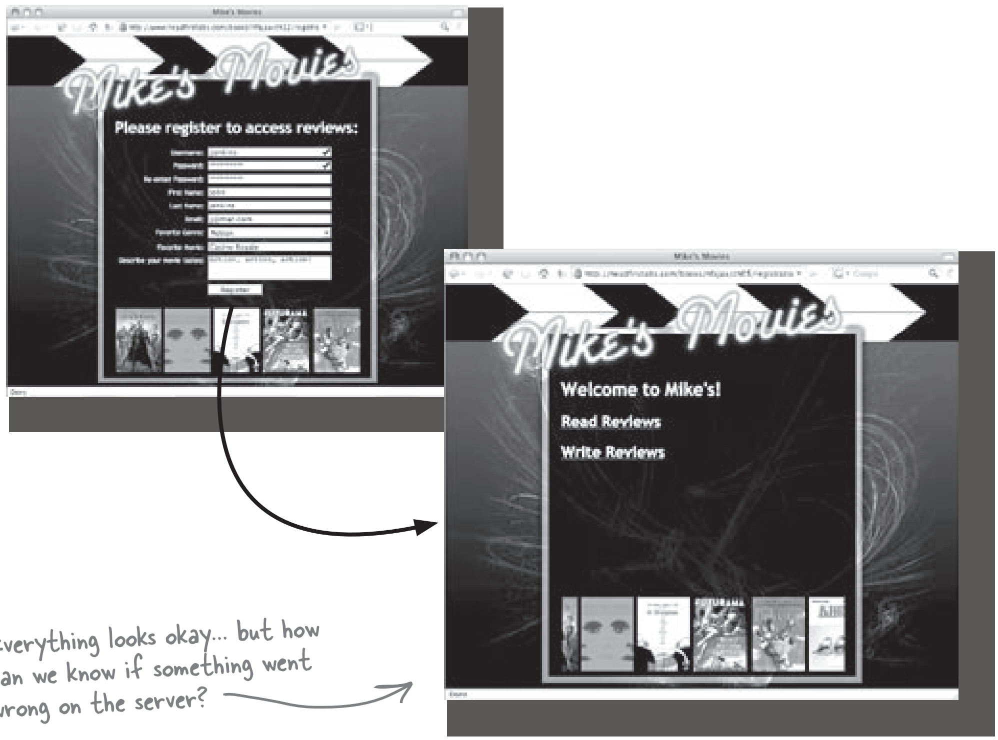
Q: But I still do need an ampersand?

A: Yes. The ampersand (&) separates different pieces of data. That tells the server where one name/value pair ends, and where the next one starts.

Test Drive

Secure Mike's app with a POST request.

Change your version of `registerUser()` in `validation.js` to match the version on page 454. Then reload Mike's registration page, and enter in some data. Try and submit the registration... does everything work like it should?



Everything looks okay... but how can we know if something went wrong on the server?

Brain Power

Would any of the other requests on the registration page be good candidates for POST requests?

Always check to make sure your request data was RECEIVED.

It seems like we're sending a valid POST request, and we know that the request data's right from when we built Mike's registration page using GET. But we really don't know for sure that our request is getting handled.

In cases like this, where you don't get direct feedback from a server, you need to check that your request data got sent to the server and was properly received. Otherwise, you could find out there's a problem much later. And problems like that are hard to debug... who remembers the code they wrote three months ago, anyway?



Good server-side programs CONFIRM the data you sent.

The server-side programs that verified usernames and passwords gave you direct feedback. That made it easy to confirm that your request data was received. In fact, most server-side programs respond to your request data and give you some sort of feedback.

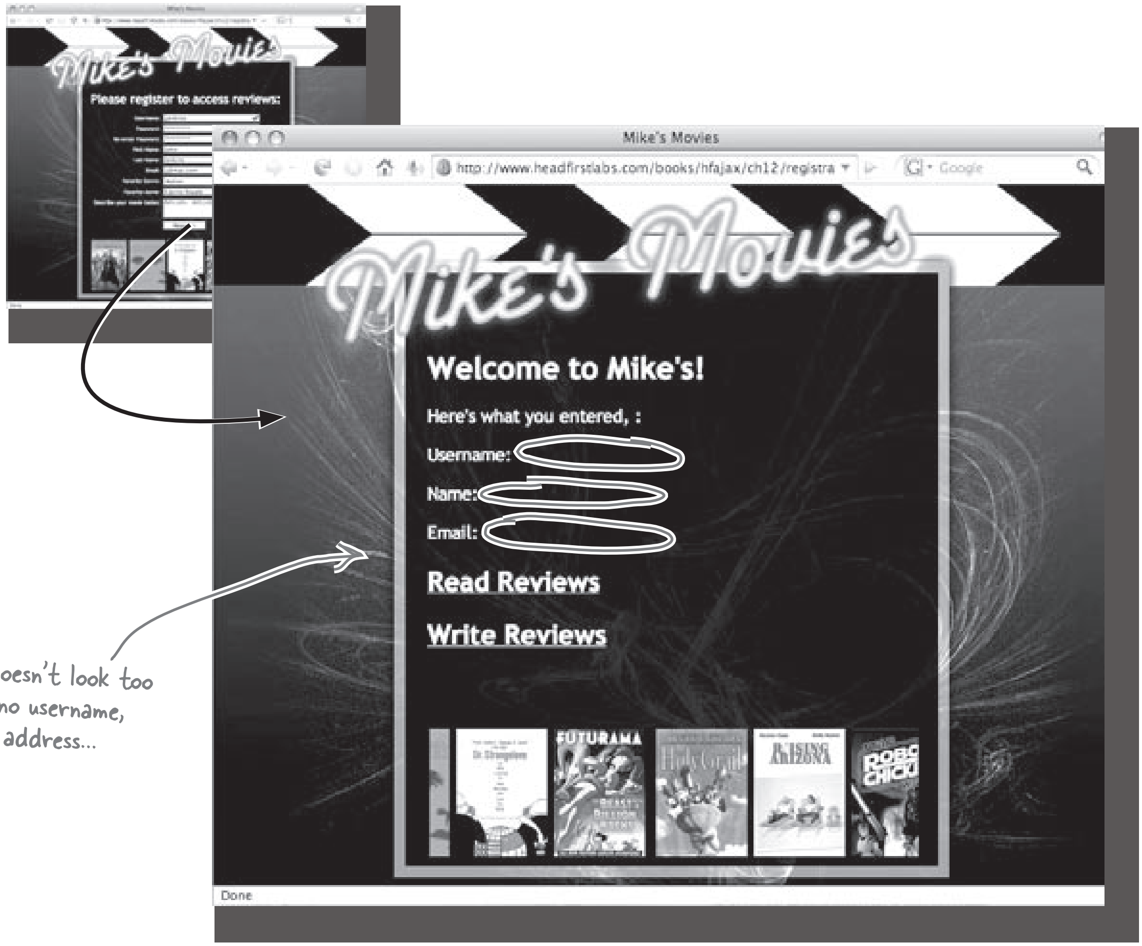
But a few programs—like Mike's server-side registration page—don't let you know what data they've received. Work with the programmers writing those programs. Often, it's easy to add a few lines and at least echo back what request data was received. Then, you can ensure the data you sent is the data that those programs received.

Test Drive (again)

Let's see what the SERVER says.

If you haven't already, download the example files for Chapter 12 from Head First Labs. There's an updated version of `register.php` called `register-feedback.php` that gives you some visual feedback when a new user submits their registration data.

Update the request URL in `registerUser()`, in `validation.js`, to use this new script. Then, try Mike's registration page again.



Uh oh... this doesn't look too good. There's no username, name, or email address...

Why didn't the POST request work?



Jim

It's got to be the register.php script. We did everything right on our end, so it's got to be a server problem.

I don't think so... those server-side guys say nothing's wrong with their end.



This is Jill... she's been hanging out with Mike's server-side guys lately.

Jim: Are you sure? I'll bet someone forgot to change the script to accept POST parameters. Come on, that's got to be it! Fix the thing, and we can get on with it...

Jill: No, I asked him about that specifically. The script accepts GET and POST parameters. Are you sure you sent the customer's details over?

Jim: I'm positive. `registerUser()` uses a POST request, and I know the request object works from when I was still using GET.

Jill: Well, you must have made a mistake somewhere.

Jim: No way. All the data's in the `send()` method of my request object... I even double-checked. So I know the data's getting to the web server.

Jill: Well, it's not getting to the script. Look at the output page! There's nothing for username, firstname, or lastname, or anything.

Jim: Wait a second. If I'm sending the data to the server correctly...

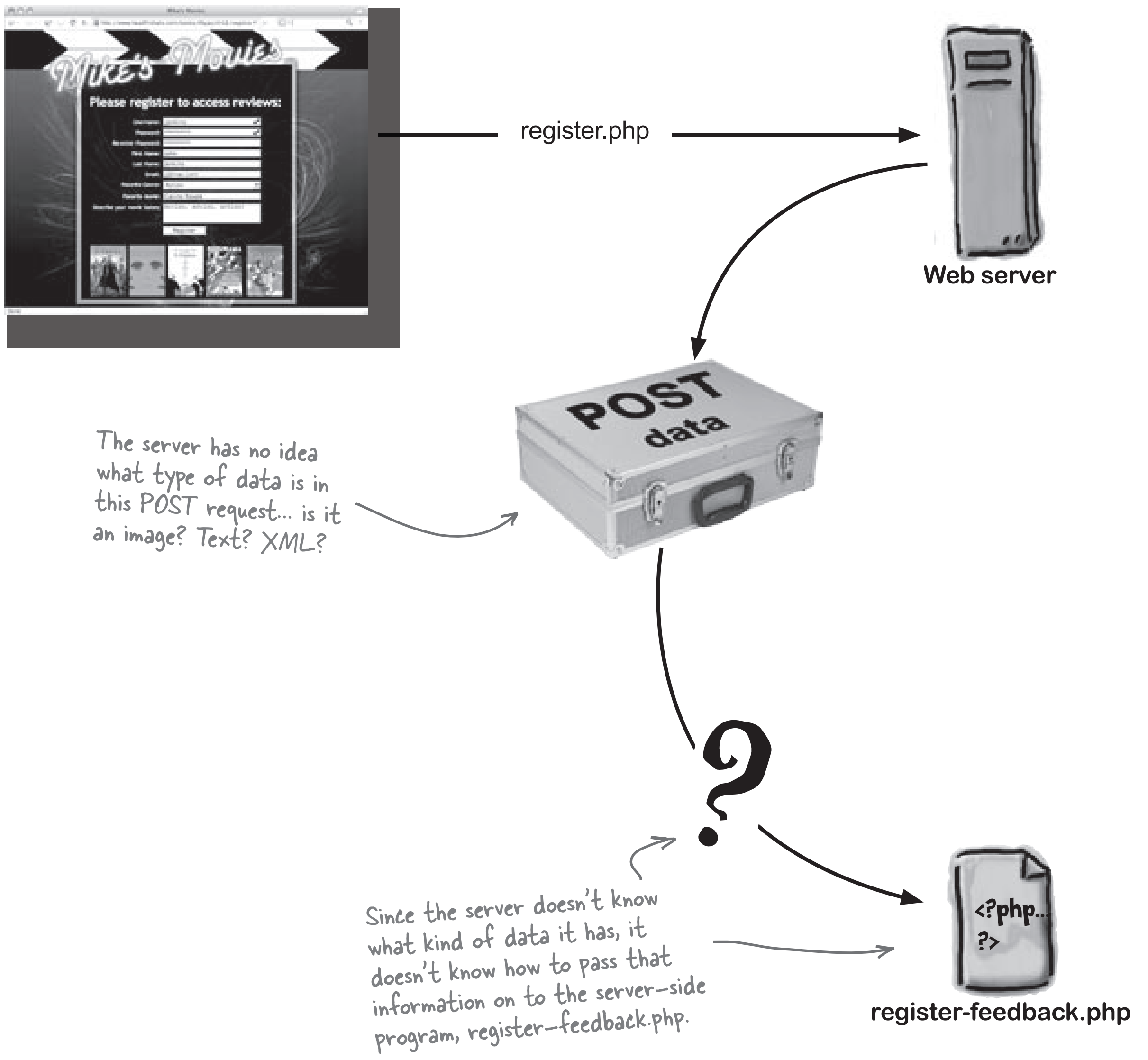
Jill: ...and the script's asking the server for the data and getting nothing...

Together: *The problem must be the server!*

The server unencodes POST data

Our script is sending a request to the server with the right request data. But somehow, the server's not getting that data to the server-side program, `register-feedback.php`. So what's going on between the server and `register-feedback.php`?

We know the server is supposed to take our POST data and unencode it. But the server has to know ***how*** to unencode that data... and that means knowing what ***type*** of data it's receiving.

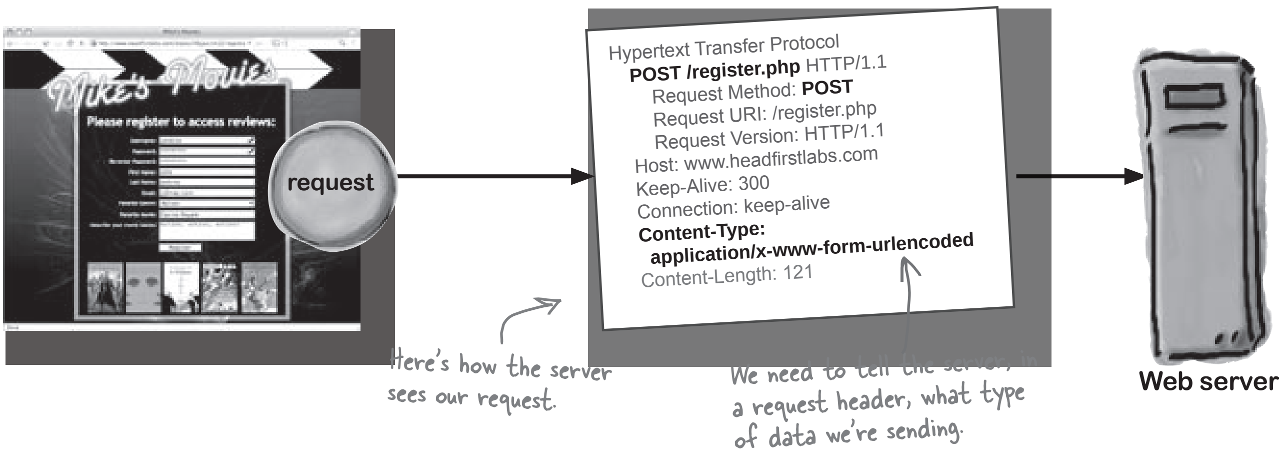


We need to TELL the server what we're sending

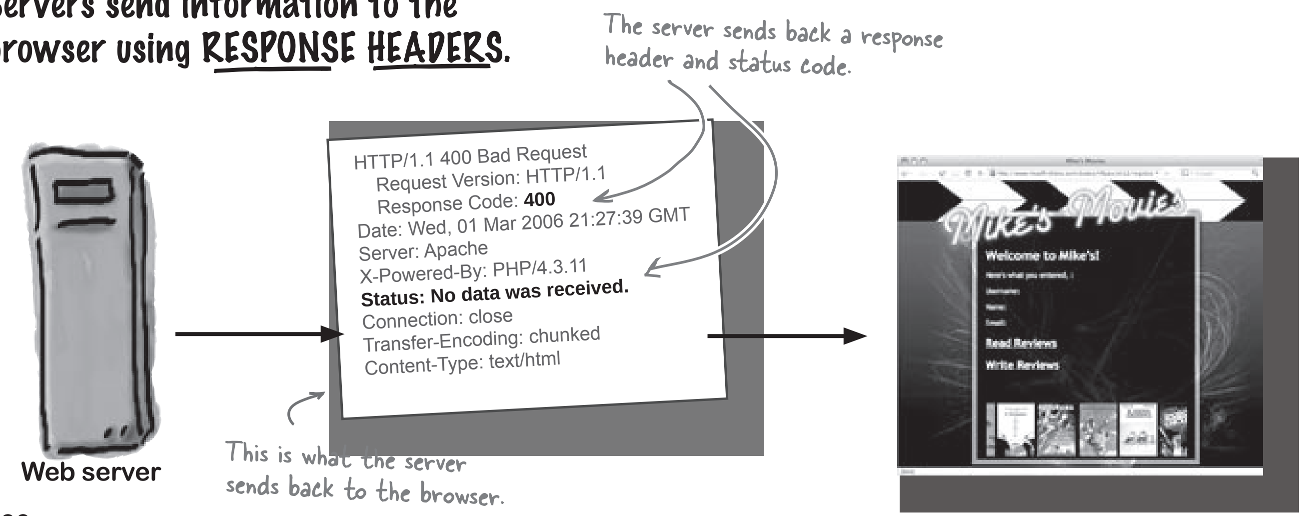
We need to let the server know exactly what type of data we're sending it. But that information can't be part of the request data itself, so we need another way to tell the server.

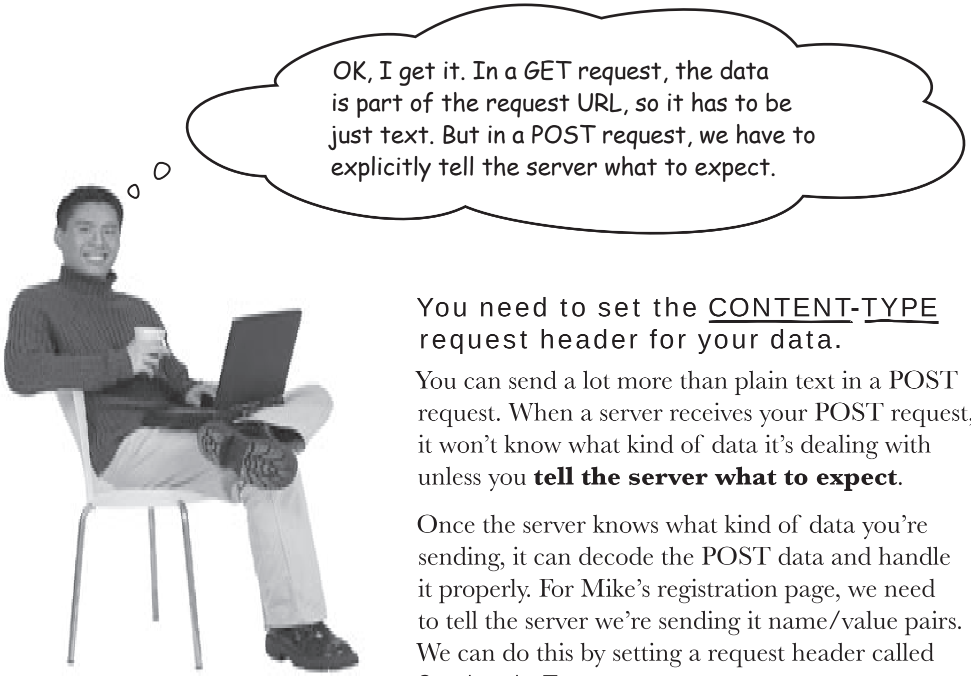
Anytime you need to talk to the server about a request, you use a **request header**. A request header is information that's sent along with the request and the server can read right away. The server can also send back **response headers**, which are pieces of information about that server's response:

Servers get information from the browser via REQUEST HEADERS.



Servers send information to the browser using RESPONSE HEADERS.

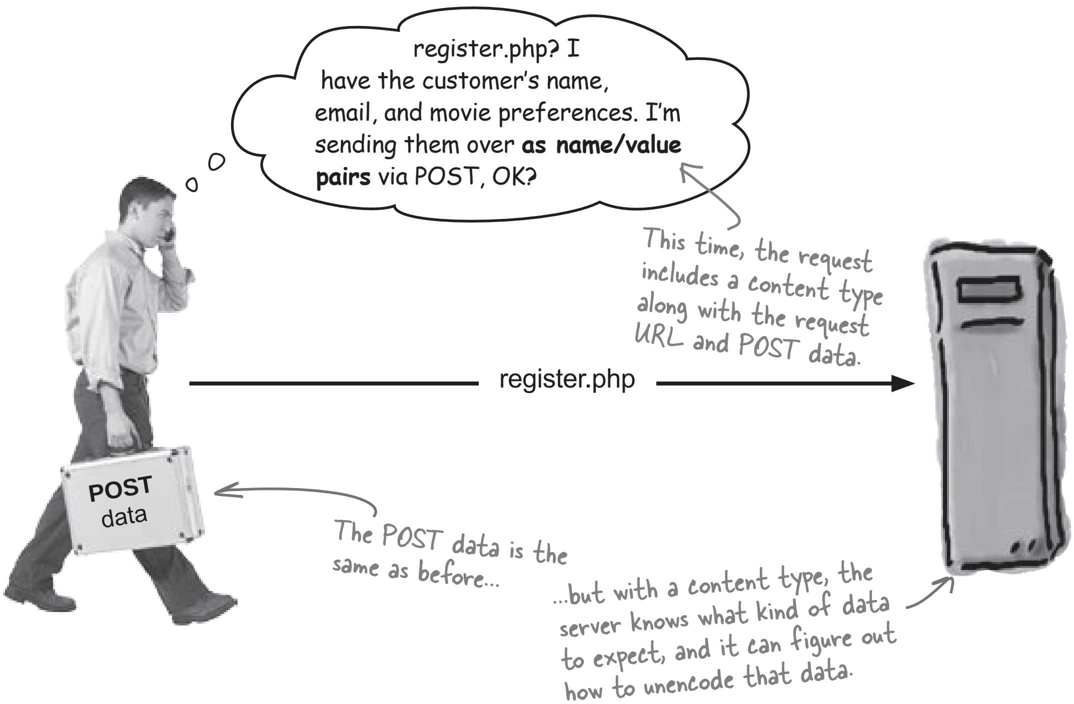




You need to set the CONTENT-TYPE request header for your data.

You can send a lot more than plain text in a POST request. When a server receives your POST request, it won't know what kind of data it's dealing with unless you **tell the server what to expect**.

Once the server knows what kind of data you're sending, it can decode the POST data and handle it properly. For Mike's registration page, we need to tell the server we're sending it name/value pairs. We can do this by setting a request header called `Content-Type`.



Set a request header using setRequestHeader() on your request object

Once you know what request header to set, it's easy to do. Just call `setRequestHeader()` on your request object, and pass in the name of the request header and the value for that header.

For name/value pairs, we want to set the `Content-Type` request header. We need to set the value of that header as `application/x-www-form-urlencoded`. That's a bit of a strange string, but it just tells the server we're sending it name/value pairs, like a web form would send:

```
function registerUser() {
  t = setInterval("scrollImages()", 50);
  document.getElementById("register").value = "Processing...";
  registerRequest = createRequest();
  if (registerRequest == null) {
    alert("Unable to create request.");
  } else {
    var url = "register.php";
    var requestData = "username=" +
      escape(document.getElementById("username").value) + "&password=" +
      escape(document.getElementById("password1").value) + "&firstname=" +
      escape(document.getElementById("firstname").value) + "&lastname=" +
      escape(document.getElementById("lastname").value) + "&email=" +
      escape(document.getElementById("email").value) + "&genre=" +
      escape(document.getElementById("genre").value) + "&favorite=" +
      escape(document.getElementById("favorite").value) + "&tastes=" +
      escape(document.getElementById("tastes").value);
    registerRequest.onreadystatechange = registrationProcessed;
    registerRequest.open("POST", url, true);
    registerRequest.setRequestHeader("Content-Type",
      "application/x-www-form-urlencoded");
    registerRequest.send(requestData);
  }
}
```

This sets the Content-Type request header...

...and tells the server to expect name/value pairs, like a web form would send in a submission.

there are no Dumb Questions

Q: So a request header is sent to the server along with the request?

A: Yes. All request headers are part of the request. In fact, the browser sets some request headers automatically, so you're really just adding a request header to the existing ones.

Q: Have we been getting response headers all along, too?

A: Yup. The browser and server always generate headers. You only have to worry about them if there's information you need to work with, like setting the content type or retrieving a status from a response header.

Q: So "Content-Type" is used to tell the server what kind of POST data we're sending?

A: Exactly. In this case, we're using name/value pairs, and the content type for that is "application/x-www-form-urlencoded." That particular type tells the server to look for values like those it would get from a normal form submission.

Q: Are there other content types?

A: Tons. To find out about the rest of them, try searching for "HTTP Content-Type" in your favorite search engine.



BRAIN BARBELL

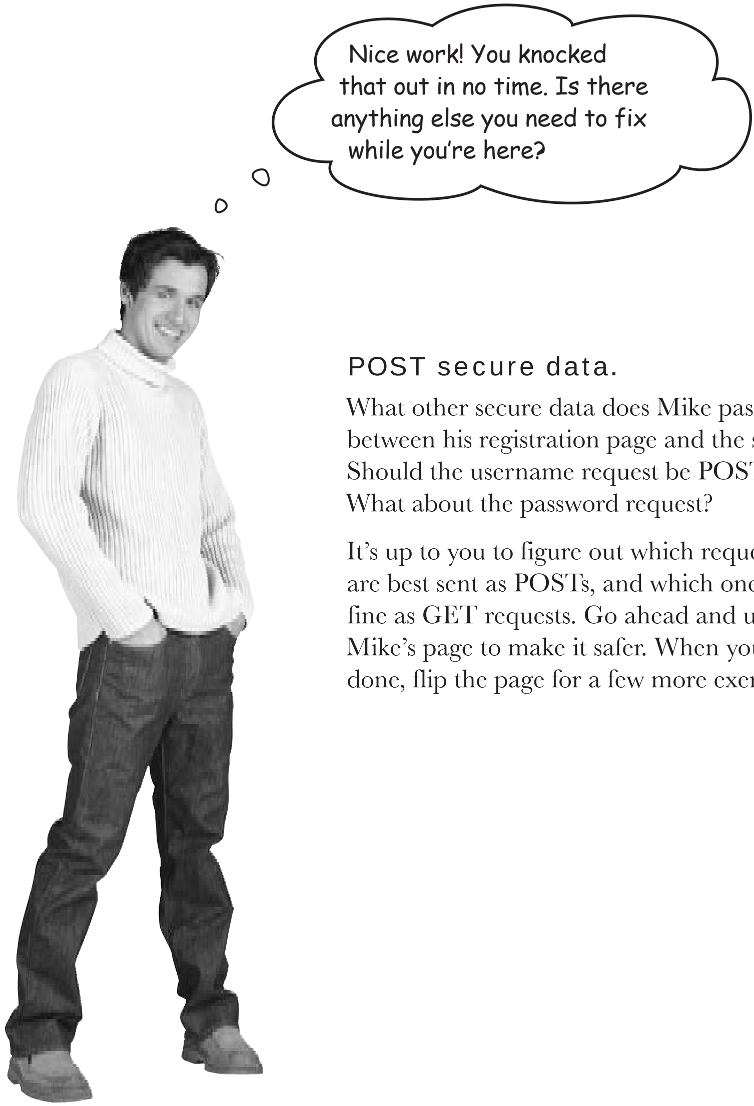
Suppose you wanted to send XML data to a server-side program. What do you think you'd need to do in order for the web server to unencode that data properly?

Test Drive (one more time)

Did it work? Did it work?

Update your request to include a `Content-Type` request header, and try Mike's registration page again. Submit your information, and see what the server says.

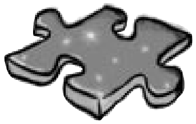




POST secure data.

What other secure data does Mike pass between his registration page and the server? Should the username request be POST? What about the password request?

It's up to you to figure out which requests are best sent as POSTs, and which ones are fine as GET requests. Go ahead and update Mike's page to make it safer. When you're done, flip the page for a few more exercises.



Word Search

X	A	R	S	Y	R	O	T	A	D	N	A	M
A	C	L	V	V	R	E	T	N	I	T	E	S
A	V	I	O	A	S	B	A	L	T	R	S	V
Q	S	L	X	H	L	N	D	L	E	R	S	L
C	U	Y	O	R	S	I	A	E	A	Y	A	R
A	C	N	N	E	U	T	D	Y	N	S	R	A
L	O	E	C	C	B	T	U	A	D	N	E	S
L	P	U	K	A	M	A	N	N	T	O	L	N
G	R	Y	C	C	I	S	E	O	X	I	B	R
E	N	I	A	H	T	E	N	A	U	T	O	R
T	U	N	B	A	D	Q	C	N	R	P	A	N
K	N	G	T	F	A	P	O	S	T	O	S	A
N	D	U	L	R	I	E	D	R	I	U	D	Y
A	S	E	R	E	D	A	E	H	T	E	S	D
J	E	R	C	I	C	T	H	R	I	Z	A	R

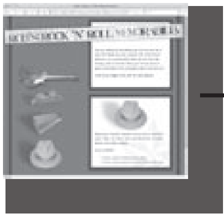
Word list:

Get
Post
Validation
Submit
Mandatory
Options
Secure
Unencode
Header
Status

GET or POST?

It's time for another episode of "GET or POST?" It's up to you to decide which request method is best for each of the following web apps.

GET or POST



Login to see my favorite rock items



Request today's house blend



Update journal with new entry



Enroll in an Advanced Yoga class



Buy "Push" from iTunes with my credit card

Word Search Solution

X	A	R	S	Y	R	O	T	A	D	N	A	M
A	C	L	V	V	R	E	T	N	I	T	E	S
A	V	I	O	A	S	B	A	L	T	R	S	V
Q	S	L	X	H	L	N	D	L	E	R	S	L
C	U	Y	O	R	S	I	A	E	A	Y	A	R
A	C	N	N	E	U	T	D	Y	N	S	R	A
L	O	E	C	C	B	T	U	A	D	N	E	S
L	P	U	K	A	M	A	N	N	T	O	L	N
G	R	Y	C	C	I	S	E	O	X	I	B	R
E	N	I	A	H	T	E	N	A	U	T	O	R
T	U	N	B	A	D	Q	C	N	R	P	A	N
K	N	G	T	F	A	P	O	S	T	O	S	A
N	D	U	L	R	I	E	D	R	I	U	D	Y
A	S	E	R	E	D	A	E	H	T	E	S	D
J	E	R	C	I	C	T	H	R	I	Z	A	R

Word list:

Get
Post
Validation
Submit
Mandatory
Options
Secure
Unencode
Header
Status

GET OR POST? SOLUTIONS

It's time for another episode of "GET or POST?" It's up to you to decide which request method is best for each of the following web apps.

Web app	GET	POST	Notes
 Login to see my favorite rock items		✓	Logging in usually involves a username and password—you want to secure that sort of information.
 Request today's house blend	✓		There's no need to use POST for a simple item request.
 Update journal with new entry	✓	✓	This might go either way. Are there user credentials being sent? Is the entry public or private?
 Enroll in an Advanced Yoga class		✓	Marcy asks for emails... we don't want those getting out to anyone malicious.
 Buy "Push" from iTunes with my credit card		✓	Sending credit card info requires POST and something more secure, like SSL.

appendix i: leftovers

The Top Five Topics (we didn't cover)



It's been a long ride... and you're almost to the end.

We can barely stand to let you go, but before you do, there's still a few things left to cover. We can't possibly fit everything about Ajax into one 600-page book. Well, actually, we *tried* that... but marketing felt that a 28-pound technical book wouldn't do too well on the shelves. So we threw out everything you didn't really need to know and kept the last few important bits in this appendix.

That's right... this really is the end. Well, except for one more short appendix... and the index (truly, a must-read). Oh, and a few ads in the back... well, you get the idea.

#1 Inspecting the DOM

By now, you're a pro at using the Document Object Model to update your web pages on the fly. But once you've used the DOM to make changes to your page, how can you see exactly what the web browser sees? The answer is to use a DOM inspector:



To open the DOM Inspector in Firefox, go to Tools and select DOM Inspector.

You can expand the DOM tree and click on a node to get details about that particular node in the DOM tree.

Watch it!

You have to request that the DOM inspector be installed on Windows.

When you're installing Firefox, select Custom Install, and then Web Developer Tools. That will get the DOM inspector running on Windows machines.

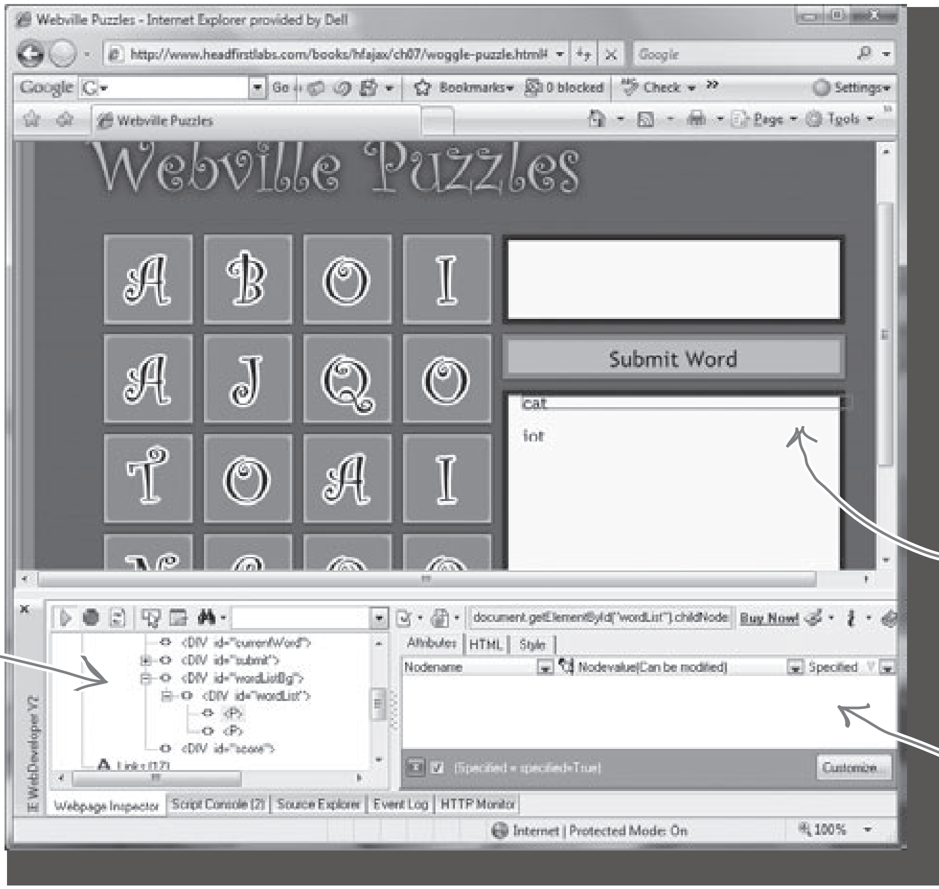
Inspecting the DOM in Internet Explorer

You'll need to download and install a separate tool for inspecting the DOM on Windows, using Internet Explorer. IEInspector is a tool that includes a DOM inspector for Internet Explorer as well as an HTTP Analyzer tool). Here's exactly what you need to do:.

Where to get it: http://www.ieinspector.com/dominspector/

How to use it: Download and install the .EXE for IEInspector. You can then launch the tool and view a page, its markup, and its DOM tree all in a single window.

IE WebDeveloper is shareware. It will cost $59 to purchase, but it's got a lot more than just DOM inspection and comes with a 30-day money-back guarantee.



Each node in the DOM can be selected below, and that node gets highlighted on the screen.

This pane displays the DOM tree for the current state of the page.

Details about each node are shown here... and you can change those details on the fly!

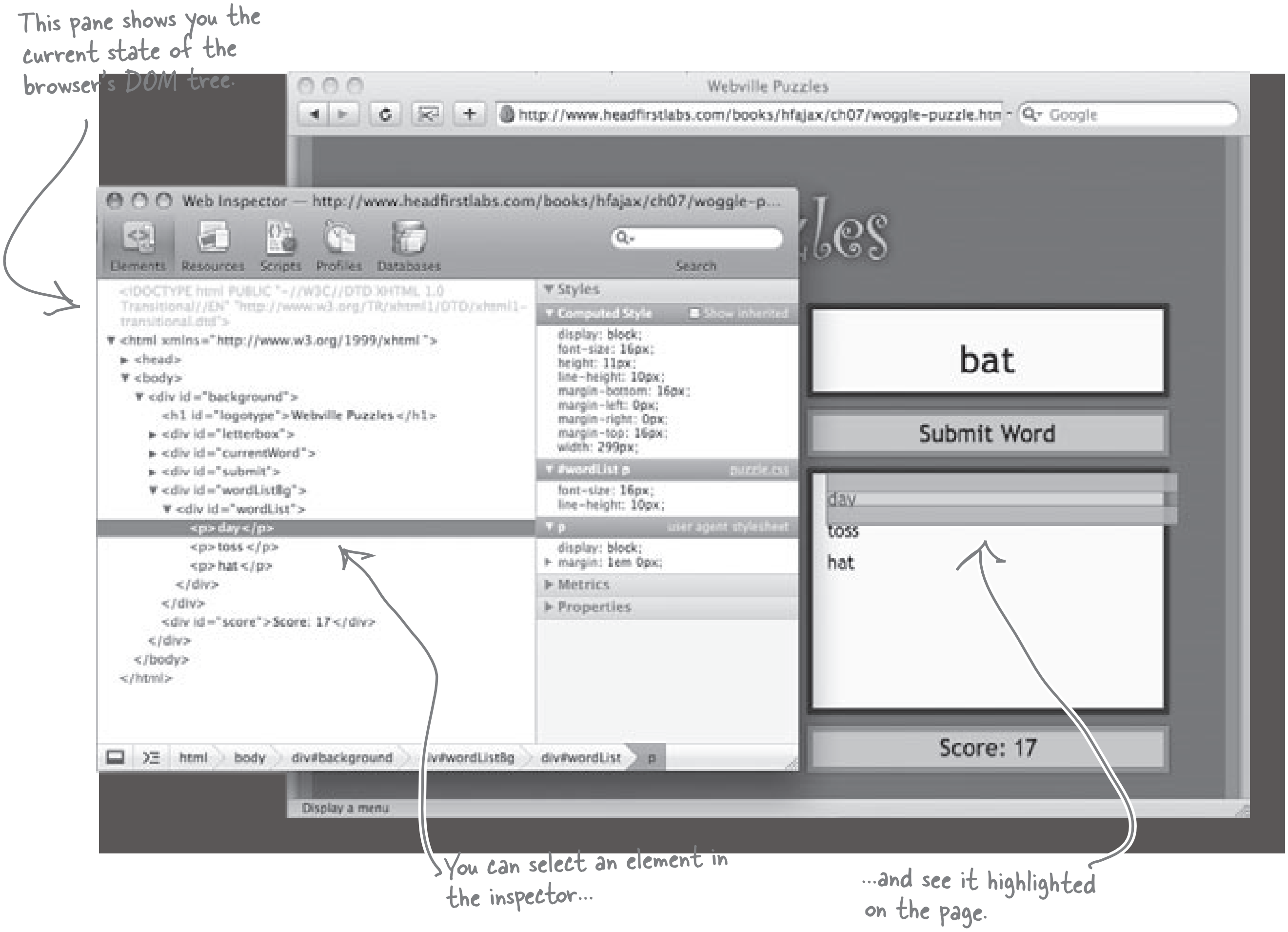
Inspecting the DOM in Safari

To inspect the DOM in Safari, you'll need to use WebKit. WebKit is the open-source system framework used by Mac OS X apps like Safari, Dashboard, and Mail. You can get it from http://webkit.opendarwin.org/.

Once you've downloaded WebKit, drag it into your Applications folder. Then you'll need to run the following command in a terminal window:

```
defaults write com.apple.Safari WebKitDeveloperExtras -bool true
```

Now go to your Applications folder and open WebKit. Right-click anywhere on your page, and select "Inspect Element:"



#2 Graceful degradation

One of the thorniest issues in Ajax is **graceful degradation**: how do you ensure that your app works for browsers *without* JavaScript enabled (or with a *really* old version of JavaScript). Since this is such a tricky topic, and lots of folks are designing apps targeted specifically at modern browsers, we've had to leave the topic for an appendix.

Still, if you're interested in creating a user experience for every possible visitor, here's what you'll need to do:

1. **Start out by designing a JavaScript-free site.**
   This is the biggest difference in building degradable sites. You can't start with an Ajaxian site and then create code to "fall back" to a non-Ajax site; there's no code that runs on a non-JavaScript browser.

   So you've got to create a site that works well ***without any JavaScript***. This is why most designers simply ignore non-JavaScript browsers.

This isn't a great way to please a really wide customer-base, but we can understand why most people go that route.

2. **Use <a> elements and Submit buttons liberally.**
   No JavaScript means no event handlers. When you're stripped of `onBlur` and `onClick`, the only markup you can use to trigger action is an `<a>` element (a link) and a form submit button. You should use those elements a *lot* since they're the only way to initiate server-side processing from a non-JavaScript page.

Here's another reason to use <a> elements for any sort of links in your page.

3. **Write server-side programs that don't assume Ajaxian requests.**
   Writing a server-side program that responds to an asynchronous request isn't fundamentally different from writing a program that responds to a form submit. The big difference, though, is in what that server-side program *returns*. The response to a validation request from an Ajax page might be "okay"; but how can a non-JavaScript page interpret "okay"?

   Instead, your server-side programs usually need to figure out, based on request parameters, what's making a request. Then, based on that, your programs will need to return different data. So for an Ajax request, your program might return a short response; for a non-Ajax request, the response might be a new XHTML page or redirect.

4. **Test, test, test... and then test some more.**
   The biggest issue in graceful degradation is testing. Even if you build a non-JavaScript version of your page and use the right elements and server-side programs, you've got to test your pages on every browser you can think of. In particular, once you've added in an Ajax version of the page, test on those non-JavaScript browsers a few more times. You never know what's crept in as you've added interactivity.

#3 script.aculo.us and the Yahoo UI libraries

You've already seen a bit about some of the cool Ajax toolkits and frameworks out there, and we mentioned script.aculo.us as one of those. Really, though, script.aculo.us is more of a user interface (UI) toolkit. There are more UI libraries out there, too. All of them are focused on making it easy to build nice, user-friendly, and sometimes visually amazing user interfaces.

These libraries are usually just JavaScript files you can download and then reference in your XHTML pages, whether the page is connected to asynchronous JavaScript or not.

script.aculo.us

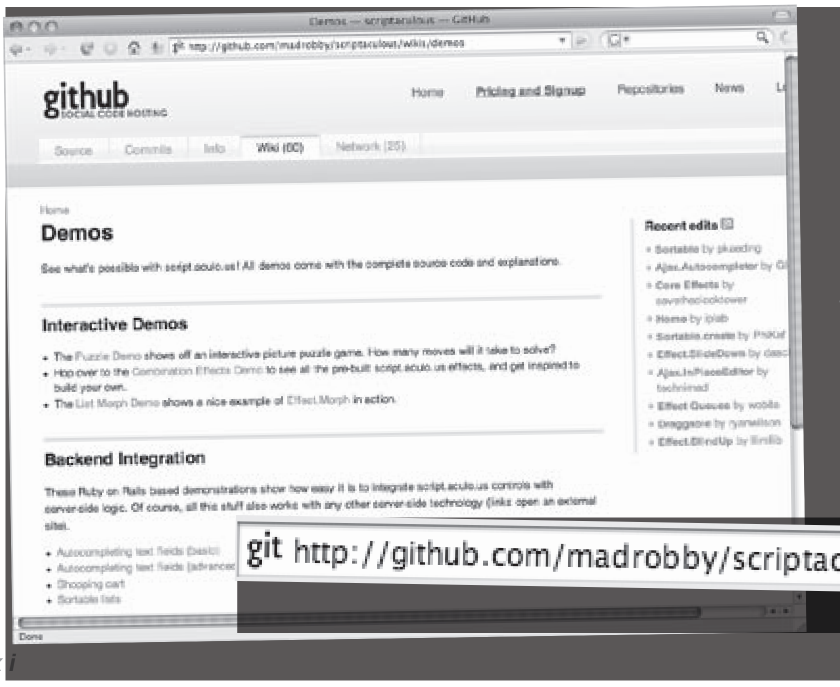
Where to get it: http://script.aculo.us/

How to use it:

```
<head>
  <title>Webville Puzzles</title>
  <link rel="stylesheet" href="css/puzzle.css" type="text/css" />
  <script type="text/javascript" src="http://script.aculo.us/prototype.js"></script>
  <script type="text/javascript" src="http://script.aculo.us/scriptaculous.js"></script>
  <script type="text/javascript" src="utils.js"> </script>
  ... etc ...
```

You can even use script.aculo.us without downloading anything... just reference these URLs.

script.aculo.us uses the Prototype library for its server interaction and some lower-level JavaScript functions.



script.aculo.us has several JavaScript libraries and tons of cool effects. To learn more, check out their demos page at:

Yahoo UI (YUI)

Where to get it: http://developer.yahoo.com/yui/

How to use it:

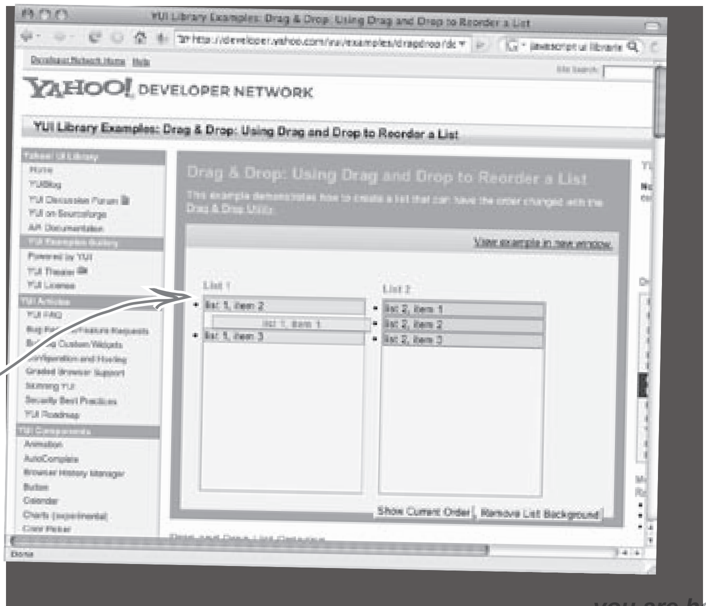
YUI breaks things up into lots of different scripts... you would probably only use a few of these for most pages.

```
<head>
 <title>Webville Puzzles</title>
 <link rel="stylesheet" href="css/puzzle.css" type="text/css" />
 <script type="text/javascript"
    src="http://yui.yahooapis.com/2.5.2/build/yuiloader/yuiloader-beta-min.js"></script>
 <script type="text/javascript"
    src="http://yui.yahooapis.com/2.5.2/build/dom/dom-min.js"></script>
 <script type="text/javascript"
    src="http://yui.yahooapis.com/2.5.2/build/event/event-min.js"></script>
 <script type="text/javascript"
    src="http://yui.yahooapis.com/2.5.2/build/animation/animation-min.js"></script>
 <script type="text/javascript"
    src="http://yui.yahooapis.com/2.5.2/build/dragdrop/dragdrop-min.js"></script>
<script type="text/javascript"
    src="http://yui.yahooapis.com/2.5.2/build/element/element-beta-min.js"></script>
<script type="text/javascript" src="http://yui.yahooapis.com/2.5.2/build/button/button-
min.js"></script>
 <script type="text/javascript" src="utils.js"> </script>
 ... etc ...
```

Each script does one thing: drag-and-drop, or DOM, or event handling.

Like script.aculo.us, YUI has lots of great UI features, notably some really cool drag-and-drop functionality.

Here, we're dragging an item from the top of the list to a lower position... all without page reloading or server interaction!



#4 Using JSON libraries in your PHP code

You've already seen how JSON can help you send and receive complex objects in your Java apps. But for PHP scripts, you're going to need a library if you don't want to type in your JSON-formatted data manually. That's a pain, so having a JSON library is a big deal for server-side JSON interaction.

Here's how you can use JSON in your PHP scripts, without getting into lots of JSON-specific syntax in your PHP:

Where to get it: It's included with PHP 5.2.0 and later

How to use it: Call **json_encode()** and pass in your PHP variables and data.

JSON support is now part of PHP, so you don't have to do anything to access JSON-specific functions in your PHP scripts.

You can create arrays and variables just as you normally would in your PHP code.

```
$itemGuitar = array(
  'id' => 'itemGuitar',
  'description' => 'Pete Townshend once played this
        guitar while his own axe was in the shop having
        bits of drumkit removed from it.',
  'price' => 5695.99,
  'urls' => array('http://www.thewho.com/',
                  'http://en.wikipedia.org/wiki/Pete_Townshend')
);

$output = json_encode($itemGuitar);
print($output);
```

All you need to do to convert your PHP data into JSON is call json_encode() and pass in the PHP variables or data.

Using JSON in PHP 5.1 and earlier

Say you're not using PHP 5.2, and you can't get your system upgraded (or don't want to). In that case, you'll need to download a library to get PHP working with JSON on your server.

This library is robust, easy to use, and packaged up through PEAR for easy access.

Where to get it: http://pear.php.net/pepr/pepr-proposal-show.php?id=198

How to use it: Create a new **Services_JSON** object, and pass your PHP data to the **encode()** method on that object.

You'll need to require() or require_once() the downloaded library, since it's not included as part of PHP automatically.

First, create a new object of type Services_JSON.

Your PHP data is created normally. Nothing JSON-specific here.

Finally, call encode() on your data, using the Services_JSON object you created earlier.

```
require_once('JSON.php');
$json = new Services_JSON();

$itemGuitar = array(
  'id' => 'itemGuitar',
  'description' => 'Pete Townshend once played this
         guitar while his own axe was in the shop having
         bits of drumkit removed from it.',
  'price' => 5695.99,
  'urls' => array('http://www.thewho.com/',
                  'http://en.wikipedia.org/wiki/Pete_Townshend')
);

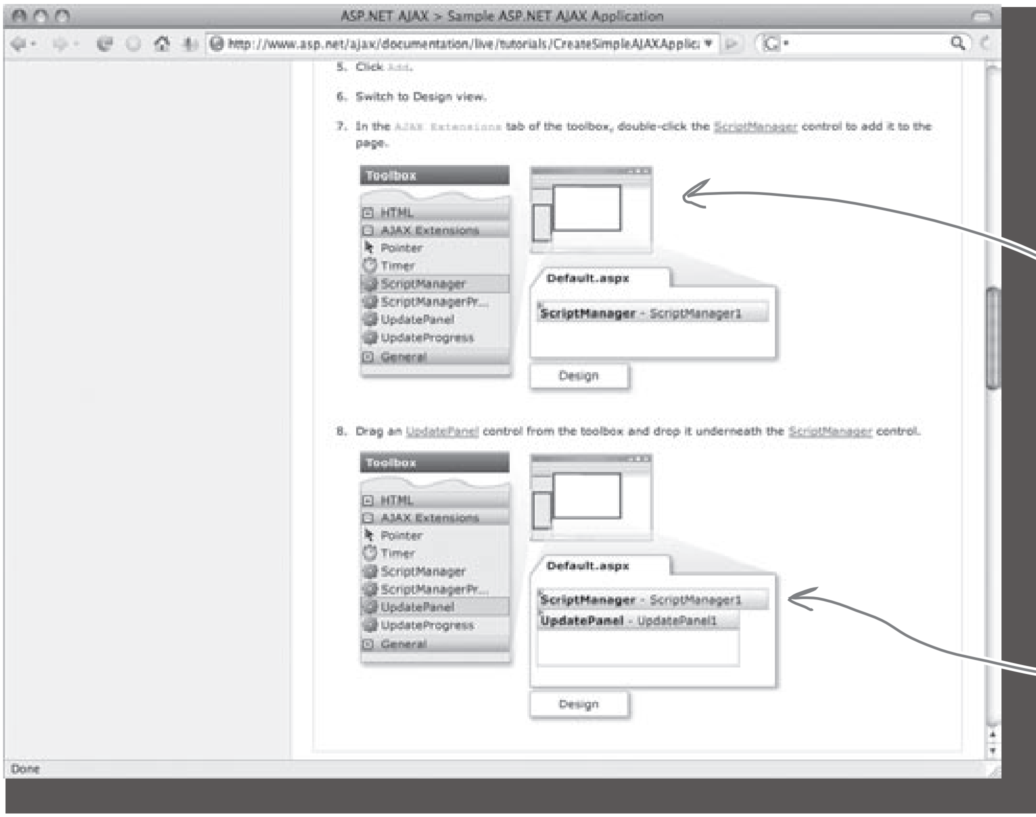
$output = $json->encode($itemGuitar);
print($output);
```

#5 Ajax and ASP.NET

If you're frequently working with Microsoft technologies, you may want to look into ASP.NET Ajax. ASP.NET Ajax is Microsoft's free, proprietary version of Ajax that hooks into Visual Studio 2008 and the rest of the Microsoft technology stack.

Because ASP.NET Ajax is built to work with Microsoft's visual products, it's more of a drag-and-drop set of front end controls, along with the ability to build code to "back" those controls.

You can find out all you ever wanted to know about ASP.NET Ajax at http://www.asp.net/ajax.



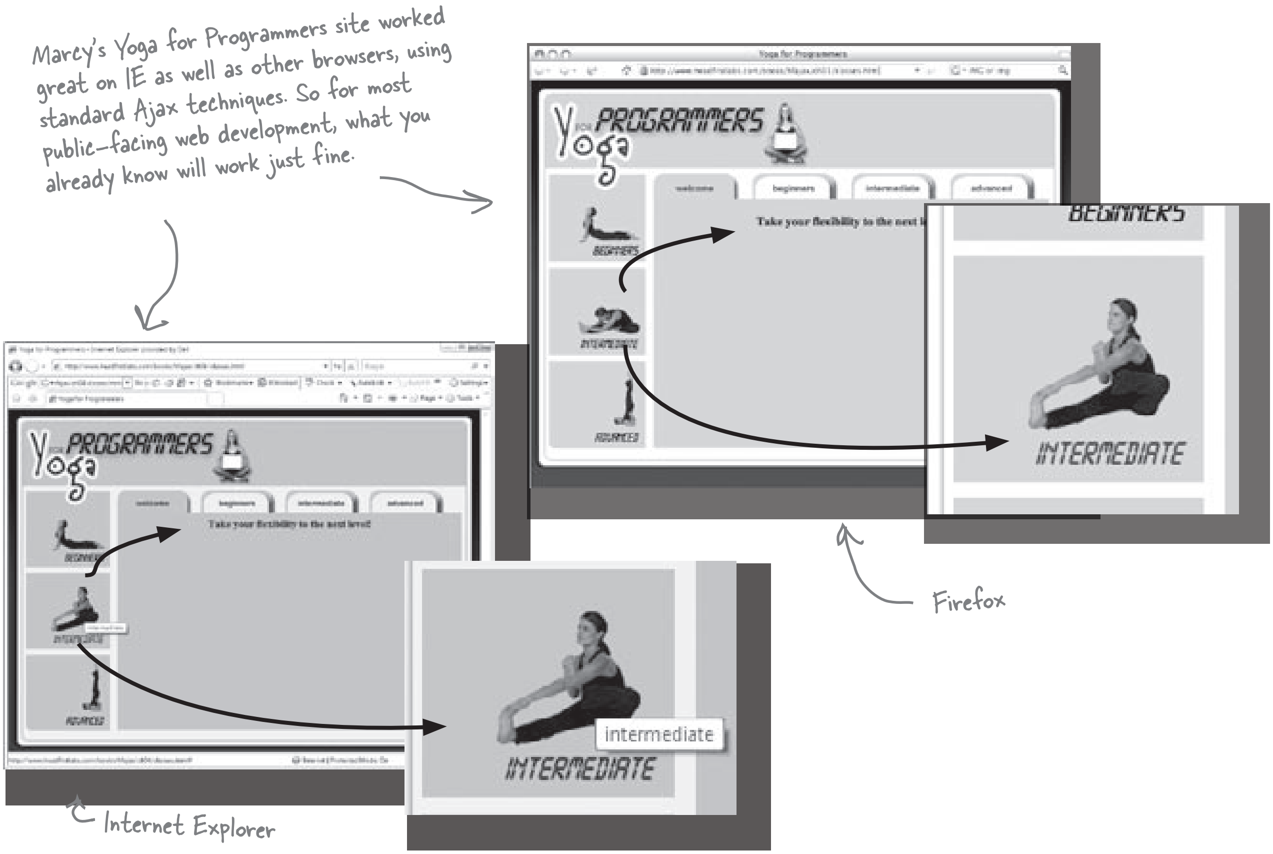
ASP.NET Ajax is more about a visual front-end for controls that interact with a server asynchronously than a new technology approach.

Microsoft has a ton of online documentation for ASP.NET Ajax... it's easy to learn a lot by just surfing the asp.net/ajax web pages.

You don't NEED ASP.NET Ajax to build Internet Explorer-compatible web apps

There are a lot of good reasons to dig into ASP.NET Ajax if you're working on or for a hardcore Microsoft product or shop. And if you're already building apps using Visual Studio, then ASP.NET Ajax will drop right in with what you're already doing.

But if you're just building web apps that you want to work on Internet Explorer—as well as other browsers like Firefox and Safari—then you don't need ASP.NET Ajax. You can use the techniques you've already learned, along with the DOM and request creation utility methods, to get your apps working on all major browsers.



appendix ii: utility functions

Just Gimme the Code



Sometimes you just want everything in one place.

You've spent a lot of time using utils.js, our little utility class of Ajax, DOM, and event utility functions. Inside these pages, you'll get all those functions in one place, ready to put into your own utility scripts and applications. Take one last chance to familiarize yourself with these functions, and then get to work on your own utilities!

utils.js: a work in progress

Here's the code we've got so far for `utils.js`. But why stop here? You should add your own functions into `utils.js`. Need a handy reusable function for getting the text out of a node? Put that function in `utils.js`.

Over time, `utils.js` should become your own personal framework.

```
function createRequest() {
  try {
    request = new XMLHttpRequest();
  } catch (tryMS) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (otherMS) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = null;
      }
    }
  }
  return request;
}

function isArray(arg) {
  if (typeof arg == 'object') {
    var criteria = arg.constructor.toString().match(/array/i);
    return (criteria != null);
  }
  return false;
}
```

This works in Safari, Firefox, Opera, etc. Everything but IE, more or less.

This is one version of Microsoft's request object...

...and here's the other.

We return null, so the calling program can figure out how to report or handle an error.

Try not to use alert() in utility functions. You need to let the function caller decide how to handle problems.

We moved isArray() into utils.js because it's a general function, not specific to a particular app.

This checks the constructor of the supplied object for the string "array," regardless of case.

If the constructor contains array, criteria won't be null, and this will return true. Otherwise, return false.

```
function getActivatedObject(e) {
  var obj;
  if (!e) {
    // early version of IE
    obj = window.event.srcElement;
  } else if (e.srcElement) {
    // IE 7 or later
    obj = e.srcElement;
  } else {
    // DOM Level 2 browser
    obj = e.target;
  }
  return obj;
}
```

This takes an Event object and returns the object that triggered that event.

Early IE stores the activated object in the window object.

Modern versions of IE use the srcElement property on the Event object.

Non-Microsoft browsers like Firefox, Opera, and Safari store the activated object in the target property of the Event object.

```
function addEventHandler(obj, eventName, handler) {
  if (document.attachEvent) {
    obj.attachEvent("on" + eventName, handler);
  } else if (document.addEventListener) {
    obj.addEventListener(eventName, handler, false);
  }
}
```

attachEvent() is Microsoft-specific syntax.

addEventListener() is for DOM Level 2 browsers.

This one's all about syntax. We need to register an event handler, but do it in a browser-independent manner.

Index

Symbols & Numbers

<a> elements 134

<script> tag 12, 50, 51, 57

404 status code 36

A

activated object 166

ActiveXObject 21
- Microsoft browsers 64

addEventHandler() 164, 165, 187
- this keyword 424–425

addEventListener( ) 142, 143, 147, 149, 150, 164
- event objects 160

addLetter( ) function 298, 308

Ajax
- benefits of 10
- design principles 47
  - expecting the unexpected 79–81
  - interactions 66
- Framework and Do-It-Myself JavaScript 332–333
- library (see frameworks and toolkits)
- request objects 398
- what is 4–5
- when to use 11

alert( ) function 58–59, 157, 428

ampersand (&) 454

appendChild( ) 255, 305

applications
- new approach 3
- traditional approach 2

arrays 394

ASP.NET Ajax 480–481

asynchronous applications 71, 173–228, 176, 187
- checkPassword() event handler 190–191
- handling events 187
- Mike's Movies 181
- monitor functions 209–213
  - callbacks 211
  - update Register button 211
  - when to call 210
- objects
  - share same name 195
  - storing callback functions 197
- one request object 196
- ordering requests and responses 203–204
- servers
  - handling events 186
  - sending a request object to the server 186
- setInterval( ) 221–222
- showPasswordStatus( ) callback 190–191
- status variables 212–213
- two request objects 198

asynchronous requests and responses 5, 8, 10–11, 46, 173–175, 197, 218–219, 314, 443

attachEvent( ) 149, 150, 164
- event objects 160

B

behavior, separating from content 82–84

benefits of Ajax 10

book examples download 14

branch 242

browser-level updates 10

browsers 65, 73
- callbacks 76
- cross-browser applications 148
- DOM trees 363
- how they see your XHTML 234

browsers (continued)
- new DOM nodes 302
- requests 10
- tree structure 236

Bullet Points
- asynchronous requests 8
- browsers 65

buttonOver( ) function 135

C

callbacks 23, 27, 33, 35, 50, 66, 76

checkPassword( ) event handler 190–191, 200–201

checkUserName( ) function 56, 70, 200–201

child 242

childNodes 308, 320, 357

child nodes, adding 255

clear text 449–450

client-side code, validating format 420–421

code
- reuse 60, 62–63
- writing large piece 296

coding violation 419

color rendering 10

compatibility issues 10

compiled languages 392

consistency 414

constraints 414

content, separating from behavior 52, 82–84

Content-Type 463
- request header 461

content versus structure 230–231

context-specific graphics 104

createAttribute( ) 303

createElement( ) method 301–302, 303

createRequest( ) function 13, 20, 22, 25, 50, 61–66, 154

createTextNode( ) 303

cross-browser
- applications 148
- utility functions 154

CSS 5, 100–101, 108
- classes 120, 134
  - images 133
- positioning 289–290
- presentation 82–84, 293

CSV (comma-separated values) 349–351, 354–357
- data 381

customers, requirements 413

D

data
- duplicate 443
- validating format and content 420–421

descriptive ids 264

design 113
- DRY: don't repeat yourself 423

disabled 21

displayDetails( ) function 13, 35, 35–36, 370–371, 395

display item's details 13

divs 12

document object...Up Close
- creating new parts of a page 233
- finding element by id attribute 233
- finding nodes by tag type 233
- getElementByID( ) 233
- getElementsByTagName( ) 233
- root element 233

DOM (document object model) 5, 143, 229–282
- appendChild( ) 255, 305
- as a tool 280
- branch 242
- changing page structure 300
- changing page without reloading 245
- child 242
- child elements 238

childNodes 262, 320
createAttribute( ) 303
createElement( ) 303
createElement( ) method 301–302
createTextNode( ) 303
dynamic application 244–245
elements
changing versus moving 252
locating by name or id 256
insertBefore( ) 305
inspecting 472–474
leaf 242
manipulating 283–328
(see also Woggle)
new nodes 302
nextSibling 262
nodeName 306–307
nodes 242, 254
adding child 255
nodeValue 306–307, 320
objects 237
parent 242
parentNode property 254, 262
parseInt 320
previousSibling 262, 320
removeChild 320
removeNode( ) 305
replaceNode 320
root element 242
text node 274, 275
trees 243, 248–251
browsers 363
Webville Puzzles 248–251
what is 232
XHTML to 234–235
xml requests and responses 359
DOM Inspector 472–474
Firefox 472
Internet Explorer 473
WebKit 474
DOM Level 0 143, 164
event handlers 158
DOM Level 1 143
DOM Level 2 143, 144, 164
Event argument 161
event handlers 158
Firefox 162
Internet Explorer 149, 162
Opera 162
Safari 162
DOM tree
family relationships 262
whitespace 272–273
XML response 360–361
Don't Annoy Your Users 47
dot notation 382
DRY (Don't Repeat Yourself) 423, 425
duplicate data 443
dynamic data 394

E

elements
changing versus moving 252
locating by name or id 256
email addresses, exposed 448
escape( ) 450
eval( ) function 385, 387–388, 392, 398, 401
warnings variable 436
Event argument 161
event framework 159
event handlers 104, 119, 140, 426–428
connecting 107
DOM Level 0 158
DOM Level 2 158
event objects 160
Internet Explorer 159
multiple 142, 424–425
new 187
properties 141
Woggle 295–296
(see also multiple event handlers)
Event Handlers Exposed 53

event listeners 144
event objects 160
events 93–138
  images 132–133
  interactivity 104
  onblur 106
  onclick 106, 107, 122
  onerror 106
  onfocus 106
  onload 106
  onmouseout 106, 122
  onmouseover 106, 107, 122
  onresize 106
  onsubmit 106
  window.onload 106, 107, 108

F

Firefox 148–150
  DOM Inspector 472
  DOM Level 2 162
  properties, methods, and behaviors 164
Fireside Chats
  Ajax Framework and Do-It-Myself JavaScript 332–333
  asynchronous and synchronous applications 227
  JSON versus XML 404
for... loop 119
forms and validation 202, 407–444
  coding violation 419
  consistency 414
  constraints 414, 416–417
  DRY (Don't Repeat Yourself) 423
  duplicate data 443
  eval( ) function
    warnings variable 436
  format and content 420–421
  multi-layered validation 414
  removing warnings 437–438
  validation process 414
  validity 414
  warnings object 433–434
form submit 28
frameworks and toolkits 99, 329–342
  jQuery 335
  mooTools 335
  pros and cons 334
  Prototype 335
  script.aculo.us 335
  syntax 336–337
  to use or not 340–341
functions, multiple 128

G

GET 22
  parameters 458
getActivatedObject( ) function 166
getDetails( ) function 13, 24–28, 32
getElementById( ) function 56, 68–69, 81, 114, 165
Get it!, XHTML fragments 128
get item's details 13
getObject( ) utility function 168–169
GET or POST? 467
  Solutions 469
GET requests 449, 461
  data in request URL 450
  GET requests versus POST requests 449
  passwords 191
  versus POST requests 453, 461
global variable 115
graceful degradation 475

H

handlers (see event handlers)
hideHint( ) 111, 117, 122
HTTP status code 36

I

ids 12
IE 5 on the Mac 61

IEInspector 473
images 124, 132–133
  CSS classes 133
initialize page 12
initPage( ) function 12, 56, 57, 113, 122, 155, 168–169
innerHTML 36, 299, 353
interactions 66, 125
interactive web pages 104
Internet Explorer 148–150
  DOM Level 2 149, 162
  event handlers 159
  IEInspector 473
  properties, methods, and behaviors 164
interpreted languages 392

J

JavaScript 5, 12, 100–101
  closing brackets 18
  defining functions 18
  disabled 21
  evaluating textual data 385
  events (see events)
  frameworks (see frameworks and toolkits)
  interpreted 392
  this keyword 158
  unobtrusive 52
jQuery 330, 335
JSON (JavaScript Standard Object Notation) 5, 379–406
  converting response to object 388
  data 382
    evaluating 386
    server's response 383
  data format 389
  eval( ) function 385, 387–388, 398
  evaluating textual data 385
  label names 397
  name-to-value mappings 430
  object properties 393
  objects 393
  PHP code 478–479
  properties 430
  property names 397
  response string 386
  Rob's Rock 'n' Roll Memorabilia
    displayDetails( ) function 395
  text and objects 381–382
  textual representation 389
  versus XML 389
JSON.parse( ) 399, 401
JSON.php 390

L

label names 397
leaf 242

M

Math.floor( ) and Math.random( ) 291, 293
Mike's Movies 44–92
  asynchronous applications 176, 181
    submit form 181–183
    update XHTML page 181–183
    validate user's passwords 181–183
  checkFormStatus( ) 209, 213
  checkPassword( ) 200–201
  checkUserName( ) 200–201
  completing registration page 224
  CSS 82–84
  one request object 196
  ordering the steps to build 50
  password verification 178
  POST requests 452
  Register button 85
  registering new event handler 219
  registration page
    allowing appropriate 85–86
    expecting the unexpected 79–81
    XHTML and CSS 51
  sending asynchronous request to server 219
  setRequestHeader( ) 462
  showPasswordStatus( ) 200–201
  showUsernameStatus( ) 200–201

Mike's Movies (continued)
- spam 446–448
- submit button 218
- Test Drive 192, 214, 225
  - with password and images 184
- window.onload event handler 54–58

monitor functions 209–213, 215
- callbacks 211
- status variables 212–213
- update Register button 211
- when to call 210

mooTools 99, 330, 335

multi-layered validation 414

MULTI-valued response 347

multiple event handlers 139–172, 142, 424–425
- addEventListener( ) 142

N

name-to-value mappings 430

nodeName 306–307

nodeValue 306–307, 320

No Dumb Questions
- <a> elements 121
- <div> 259
- 404 status code 36
- ActiveXObject 21
- addEventHandler( ) 165, 187
- addEventListener( ) 143, 150
- addLetter( ) 308
- Ajax, when to use 11
- alert( ) function 58
- ampersand (&) 454
- appendChild( ) 305
- assigning requests to other variable names 199
- asynchronous 5
- asynchronous requests 11
- asynchrony 199
- attachEvent( ) 150
- browsers 150
- callback function 33
- child elements 238
- childNodes 308, 357
- coding violation 419
- Content-Type 463
- createRequest( ) function 63, 152
- CSS 124
- CSS class 134
- CSS positioning 290
- CSV 357
- currentWordDiv.firstChild 316
- design principles 47
- disabling all fields while images are scrolling 220
- displayDetails( ) function 35
- document object 165
- DOM 143, 165, 238, 247
- DOM code 270
- DOM Level 0 143
- DOM Level 1 143
- DOM Level 2 143
- DOM tree 255
- DRY (Don't Repeat Yourself) 425
- eval( ) function 387–388, 401
- Firefox 150
- frameworks and toolkits 331, 339
- getElementById( ) function 69, 165, 259
- GET requests and passwords 191
- global functions 199
- global variables 199
- href 121
- HTTP status code 36
- innerHTML 36
- insertBefore( ) 305
- Internet Explorer (IE) 150
- Internet Explorer framework 159
- JavaScript, disabled 21
- JSON
  - data 387
  - data format 389
  - textual representation 389
- JSON.parse( ) 401
- mixing style into XHTML 183
- monitor functions 215
- multiple event handlers 425

nodes 255
onclick 165
onreadystatechange property 26, 33
Opera 150
password fields 215
passwordRequest 215
passwords 191
POST requests versus GET requests 453
question mark (?) 454
readyState property 33, 36
refactoring code 215
registration page 51
removeNode( ) 305
request headers 463
request object properties 26
request variables 199
response headers 463
restriction 202
root element 255
Safari 150
self-describing data 369
server traffic 11
setAttribute( ) 357
setInterval( ) 221
src property of an <img> 255
status property 36
synchronous requests 316
tabs 134
text node 275
this keyword 165, 425
toolkits 99, 331
try...catch block 152
TSV (tab-separated values) 357
usernameRequest 215
utils.js 63
value property 69
warnings object 434
Webville Puzzles 278
Welcome tab 130
window.onload 108
window object 58
Woggle 290, 305
XHTML 358
XHTML images 124
XMLHTTP 21
XMLHttpRequest 21, 61
XMLHttpRequest object 5
XML versus JSON 389
XML vocabularies 358

null 65

O

objects
- accessing members 393
- activated 166
- converting response to 388
- DOM 237
- JSON 381–382
- properties 393
- request 398
- responseText property 383
- share same name 195
- storing callback functions 197

onblur event 50, 106

onClick event handlers 12, 16, 24, 106, 107, 122

onerror 106

onfocus 106

onload event handler 106
- window.onload event handler 54–58

onmouseout 106, 122

onmouseover 106, 107, 122

onreadystatechange property 26, 33, 72

onresize 106

onsubmit 106

open( ) method 26

Opera
- DOM Level 2 162
- properties, methods, and behaviors 164

order 204

P

page refreshes 8

parent 242

parentNode property 254

parseInt 320

parsing server response 399

passwords
- GET requests 191
- verification 178

PHP code and JSON 478–479

POST 22
- data 451
- parameters 458

POST requests 445–470, 451, 452, 458, 461
- ampersand (&) 454
- clear text 450
- code example 454
- Content-Type 463
- encoded 452
- failure 458
- GET requests versus POST requests 449
- question mark (?) 454
- request headers 460, 463
- response headers 460, 463
- testing 456
- unencoding 459
- URL 451
- versus GET requests 453

presentation, separating from content and behavior 52, 82–84

previousSibling 320

properties
- names 397
- value as another object 430
- values 297

Prototype 330, 335

Q

question mark (?) 454

R

Ready Bake Code
- createRequest( ) function 20
- function isArray( ) 394
- function scrollImages( ) 223

readyState property 33, 36, 72, 74–76

Ready States Up Close 74–75

redesign overview 12–13

refactoring code 215

removeChild 320

removeNode( ) 305

replaceNode 320

request data 452

request headers 460–463

request objects 13, 25–26, 64–65, 127, 398
- properties 22, 26
  - onreadystatechange 72
  - readyState 72
  - responseText 72
  - responseXML 72
  - status 72
  - statusText 72
- responseText property 36
- send( ) method 28

request URL 451, 461

requirements 413

response headers 460, 463

responseText property 33, 36, 72, 383

responseXML property 33, 72

restriction 202

reuse code 60, 62–63

Rob's Rock 'n' Roll Memorabilia 2–42
- CSV 356
- design problems 8

displayDetails( ) function 370–371, 395
displaying details 13, 35–37
get item's details 13
initialize page 12
onclick event handler 16
redesign overview 12–13
request object 13
Step 1: Modify the XHTML 14–15
Step 2: Initialize the JavaScript 16–19
Step 3: Create a request object 20
Step 4: Get the item's details 22–34
Test Drive 362
JSON 390, 396
XHTML web page 12

root element 242

Run it!
book examples download 14
download CSS and graphics 183
download XHTML and CSS for Woggle 289
JSON.parse( ) 399
Mike's Movies project, validation.js 56
Rock 'n' Roll project, getDetails( ) function 24

S

Safari 148–150
DOM Level 2 162
properties, methods, and behaviors 164
WebKit 474

script.aculo.us 99, 335, 476

self-describing data 366, 369

send( ) method 28

sequence 204

server-side code 29

server-side interactions 30–31

server-side programs, validating content 420

server-side scripts 58

servers 6, 458
data
CSV (comma-separated values) 349–351, 354–357
XHTML fragment 349–351
XML 349–351
duplicate data 443
format-specific response 353
generic response 353
handling events 186
MULTI-valued response 347
page-specific response 353
parsing response 399
sending a request object to the server 186
traffic 10–11

setAttribute( ) 357

setInterval( ) 221–222

setRequestHeader( ) 462

showHint( ) 111, 114–116, 122, 168–169
title property 119

showPasswordStatus( ) 190–191, 200–201

showSchedule( ) 127

showTab( ) 111, 118, 119, 123, 127, 130

showUsernameStatus( ) function 56, 76, 77, 200–201

spam 446–448
exposed email addresses 448

split 297

srcElement 162, 164

src property 252

status property 33, 36, 72

statusText property 33, 72

status variables 212–213

string-handling utility functions 297

structure versus content 230–231

substring 297

synchronous requests 174–175, 218, 314, 316
callback functions 315

T

Table Cells Exposed 259

tabs 118–124, 134
- <a> element 121

target 164

text
- evaluating textual data 385
- JSON 381–382

this keyword 158, 165, 424–425

thumbnails.js 19

toolkits (see frameworks and toolkits)

traditional approach 2

troubleshooting 157

try...catch block 152

TSV (tab-separated values) 357

typeof operator 394

U

UI libraries 476–477

unobtrusive JavaScript 52

unwarn( ) function 437–438

URLs
- data in 450
- GET request 450
- POST requests 451
- sending data 28

usability 208, 220

utility function 166

utils.js 62–63, 63, 484–485

V

validation (see forms and validation)

validation.js 56

validity 414

var currentTab = this.title 164

verification 202

W

warnings object 433–434

warnings variable 436

Watch it!
- <script> tag 51
- DOM Inspector 472
- event listeners 144
- IE 5 on the Mac 61
- image files 86
- JSON.php 390
- "s" in method names 256

WebKit 474

web pages
- changing page without reloading 245
- changing structure 300
- design 113
- new approach 3
- related objects 237
- traditional approach 2

Webville Puzzles 244–251, 260, 265
- cellIsEmpty() 274
- Fifteen Puzzle 258
- Fifteen Puzzle Code 266–269
- swapTiles( ) 274
- Test Drive 258, 271, 277, 279
- whitespace nodes 274
- Woggle (see Woggle)
- XHTML 246

Which Data Format? Text or XML 373, 377

whitespace 272

window.onload 58, 106, 107, 108
- event handler 54–58

window.srcElement 164

window object 58

Woggle
- addLetter( ) function 298, 308
- alert( ) statement 298
- childNodes 308
- CSS presentation 293
- currentWord 300
- disable tiles 310
- download code 289
- event handler 295–296
- initPage( ) 292
- letter frequency 291
- letter recognition 298
- randomizeTiles( ) function 292, 296
- random numbers 291
- submitting words 312
- submitWord( ) function 316, 317, 322–324
- Test Drive 294, 325
  - addLetter( ) 311
  - event handlers 309
  - letter recognition 298
  - word score 318
- turning off addLetter( ) 310
- XHTML 288–289

X

XHTML 5, 100–102, 108
- <a> elements 134
- descriptive ids 264
- fragments 128, 349–351
- images 124
- innerHTML 299
- links 134
- to DOM 234–235
- web page 12

XML 5, 349–351
- JSON 381
- self-describing data 366, 369
- versus JSON 389

XMLHttpRequest 5, 21, 61
- non-Microsoft browsers 64

XML requests and responses 343–378
- DOM 359
- DOM trees 360–361
  - browsers 363
- innerHTML 353
- MULTI-valued response 347
- self-describing data 366

XML vocabularies 358

Y

Yahoo UI libraries 476–477

Yoga for Programmers
- buttonOver( ) function 135
- constraints 416–417
- createRequest( ) function 154
- CSS classes 120
- DRY (Don’t Repeat Yourself) 425
- emailIsProper( ) function 426
- event handlers 426–428
- fieldIsFilled( ) function 426
- fieldIsLetters( ) function 427
- fieldIsNumbers( ) function 427
- getElementById( ) 114
- getObject( ) utility function 168–169
- get triggered object 166
- hideHint( ) 111, 117, 122
- images 132–133
- initPage( ) function 113, 122, 155, 168–169, 428
- problems 410–412, 441–442
- request object 127
- showHint( ) 111, 114–115, 122, 168–169
- showSchedule( ) 127
- showTab( ) 111, 123, 127, 130
- tabs 118–124
- Test Drive 103, 111
  - browser testing 147, 156, 170
  - class schedule page 117, 123, 131, 136
  - error counting 418, 435
  - turning warnings on and off 439
- validation 421
- warnings object 433–434
- XHTML 102, 114–115